Neural Program Calculus

A Formal System for Specifying Intent to Self-Organizing Interpreters

MATTHEW BUSIGIN, VoxGenius, Inc., USA
GEORGI D. GOSPODINOV, Independent Researcher, USA
ARBAAZ MAHMOOD, VoxGenius, Inc., USA

We present a preliminary formalization of the "neural program" paradigm—a hybrid notation combining natural language and pseudocode used to specify complex tasks to large language models and other self-organizing AI systems. Drawing on Glushkov's framework of alphabetic operators, automata theory, and self-organizing systems, we define a calculus with explicit syntax, operational semantics, and conjectured theorems. This calculus occupies a novel position in the space of formal languages: it is interpreted not by a deterministic automaton but by a probabilistic, pattern-recognizing system that resolves intent rather than executing literal instructions. The syntax and grammar are fully deterministic; probabilistic behavior arises solely from interpreter-mediated semantic resolution.

CCS Concepts: • **Software and its engineering → Context specific languages**; *Constraint and logic languages*; • **Computing methodologies → Natural language processing**; *Neural networks*; • **Theory of computation → Semantics and reasoning**.

Additional Key Words and Phrases: neural programs, large language models, formal semantics, self-organizing systems, probabilistic programming, intent specification, Glushkov automata

Preface: On the Absurdity of This Document

There is no such thing as AGI. This is because neither intelligence nor generality is a binary classification: they are, in fact, curves over task distributions and resource constraints. Every system exhibits some degree of generality across some range of tasks and some degree of intelligence within some domain. The question is never "is it AGI?" but rather "how general?" and "how intelligent?" and "at what cost?" and "with what failure modes?"

The mystification serves no one. What practitioners need is clarity: precise vocabulary, formal foundations, and honest assessment of capabilities and limitations.

Who we are. This document is a handbook from practitioners. The authors have spent the past six years building, deploying to production, and operating progressively more general and more intelligent systems. We have shipped neural programs that process millions of transactions. We have debugged failures at 3 AM that no stack trace could explain. We have learned what these systems can and cannot do—not from papers, but from pager alerts.

This is not a theoretical exercise. It is an attempt to formalize what we have learned the hard way.

What LLMs actually are. Large language models are billion-parameter function approximators trained on internet text. They are:

- **Not reasoning engines.** They are pattern-matching systems that produce statistically plausible continuations. When they appear to reason, they are retrieving and recombining patterns that *look like* reasoning.
- **Not knowledge bases.** They are lossy compressions of training data. They confabulate. They state falsehoods with perfect confidence. They cannot distinguish what they know from what they are inventing.

Authors' Contact Information: Matthew Busigin, matt@voxgenius.ai, VoxGenius, Inc., USA; Georgi D. Gospodinov, georgi.d.gospodinov@gmail.com, Independent Researcher, USA; Arbaaz Mahmood, arbaaz@voxgenius.ai, VoxGenius, Inc., USA.

- **Not reliable.** Given the same input twice, they may produce different outputs. Given a slightly rephrased input, they may produce wildly different outputs. Their behavior is sensitive to prompt formatting, token boundaries, and other minor, effectively unmodeled perturbations, e.g. the phases of the moon.
- **Not transparent.** No one can fully explain why a particular output was produced. The internal representations are high-dimensional numerical fog. "Interpretability" is not merely difficult—it is undecidable. You cannot, in general, determine what computation a neural network is performing. Glushkov formalized this sixty years ago: self-organizing systems modify their own behavior in ways that cannot be predicted from their initial specification. This is not a funding problem. It is a theorem.

None of this prevents them from being useful. It simply means we must engineer around it.

What this document is not. This is not a claim that LLMs are intelligent, conscious, or approaching either. This is not a prediction that neural programs will replace traditional software engineering. This is not written by people who believe the hype.

We have built these systems. We know what they are.

The fundamental limitation. Section 16 of this document formalizes what we call *semantic incompleteness*: the structural inability of bounded computation to guarantee preservation of meaning under increasing semantic load. This is not a bug to be fixed. It is not a scaling problem. It is a computability constraint—and like all such constraints, it applies reflexively. This document cannot fully verify itself.

So why write it? Because the systems exist. Because people are using them. Because "prompt engineering" has become a discipline practiced by millions without any formal foundation. Because the alternative to formalization is not "no LLMs"—it is LLMs deployed without rigor, without bounds, without the vocabulary to describe their failure modes.

We do not write this because we believe LLMs are the future of computation. We write it because they are the *present* of computation, and the present deserves the same analytical treatment we would give any other tool.

What remains. You cannot accuse us of overhyping LLMs. We have called them unreliable, non-transparent, semantically incomplete pattern matchers. You cannot accuse us of ignoring failure modes. We have dedicated an entire section to formalizing why they fail and an appendix cataloging how they fail. You cannot accuse us of naive optimism. We have stated, in a peer-reviewed document, that the systems we describe *cannot* provide the guarantees that serious engineering requires. You cannot accuse us of theorizing from the sidelines. We have six years of production deployments behind us.

What remains is the work itself: a formal calculus for specifying intent to systems that resolve intent imperfectly. If the formalization is wrong, show us where. If the semantics are incomplete, extend them. If the theorems are false, prove counterexamples.

— The Authors, January 2026

Contents

Abstract 1
Contents 2

1 Introduction 7
1.1 Motivating Example 7
2 Syntax 7
2.1 Alphabet 7
2.2 Grammar 8
2.3 Syntactic Sugar 8
3 Semantic Domains 9
3.1 Values 9
3.2 Context 9
3.3 The Interpretation Function 10
4 Operational Semantics 10
4.1 Execution 10
4.2 Sequencing 10
4.3 Action Interpretation 11
4.4 Iteration 11
4.5 Delegation 12
5 Validation Semantics 13
6 Type System 13
6.1 Format Types 14
6.2 Action Signatures 14
6.3 Well-Typedness 14
7 Design Patterns as Compositional Operators 15
7.1 The Continuation Operator 15
7.2 The Compression Operator 15
7.3 The Journaling Operator 16
7.4 The Semantic Index Operator 16
7.5 The Counteragent Operator 16
7.6 The Ledger Operator 16
7.7 Operator Composition 16
8 Connection to Glushkov's Theory 17
8.1 Alphabetic Operators Revisited 17
8.2 Neural Automata 17
8.3 Neural Languages 18
8.4 Self-Organization as Context Dependence 18
8.5 Glushkov's Synthesis Realized 18
9 Probabilistic Correctness and Verification 18
9.1 Partial Correctness 18
9.2 Verification Strategies 19
9.3 Compositional Verification 19
9.4 The Halting Problem 20
9.5 Confidence Estimation 20
10 Axioms and Theorems 20
10.1 Axioms 21
10.2 Propositions 21
10.3 Conjectures 21
10.4 Scope Note 22
11 Extensions 22
11.1 Structured Error Handling 22

11.2 Quantitative Validation Criteria 22
11.3 Named Definitions and Abstraction 23
11.4 Streaming and Incremental Execution 23
11.5 Status of Extensions 24
12 Discussion 24
12.1 Related Work and Intellectual History 24
12.2 Comparison with Traditional Formalisms 26
12.3 Open Problems 26
13 Case Study: Production Counteragent System 26
13.1 System Overview 26
13.2 The Neural Program 26
13.3 Novel Techniques 28
13.4 Formal Analysis 30
13.5 Observed Properties 30
13.6 Lessons for Neural Program Design 31
14 Extended Case Study: Automated Research Paper Generation 31
14.1 Problem Statement 32
14.2 Program Architecture 32
14.3 Module 1: Victory Metrics (`metrics.npc`) 32
14.4 Module 2: Source Management (`sources.npc`) 35
14.5 Module 3: Dimensional Analysis (`analysis.npc`) 37
14.6 Module 4: Synthesis with Counteragent (`synthesis.npc`) 39
14.7 Main Program (`main.npc`) 41
14.8 Paper Generation Program (`paper-generator.npc`) 44
14.9 Execution Journal 47
14.10 Mapping to NPC Formal Constructs 49
14.11 Results and Artifacts 49
14.12 Lessons and Design Principles 50
14.13 Conclusion 51
15 Formal Invariants of Survivable Agentic Systems 51
15.1 Invariant I: Boundedness 51
15.2 Invariant II: Explicit State 51
15.3 Invariant III: Traceability 52
15.4 Invariant IV: Loss Accounting 52
15.5 Invariant V: Identity Before Similarity 52
15.6 Invariant VI: Intent-Guided Retrieval 52
15.7 Invariant VII: Separation of Proposal and Commitment 52
15.8 Invariant VIII: Adversarial Pressure 52
15.9 Invariant IX: Context Isolation 53
15.10 Invariant X: Killability 53
15.11 The Meta-Invariant 53
16 Semantic Incompleteness and Compute-Boundedness 53
16.1 The Problem of Semantic Load 53
16.2 Why Scale Does Not Resolve This 54
16.3 Compute-Boundedness 54
16.4 The Compute Hierarchy 54
16.5 The Incompleteness Theorem 56
16.6 NPC as Incompleteness Management 56

16.7 The Mature Response 57
16.8 Computational Complexity Analysis 57
17 NPC Extension: Claude Code Instantiation 58
17.1 Extended Semantic Domains 58
17.2 Tool Taxonomy 59
17.3 Tool Signatures 59
17.4 Extended Grammar 61
17.5 Operational Semantics for Tools 61
17.6 Hook System Semantics 63
17.7 Permission Model 63
17.8 Subagent Semantics 64
17.9 MCP Integration 65
17.10 Session and Context Management 65
17.11 Execution Modes 66
17.12 Skill System 66
17.13 Code Execution and Interpretation 67
17.14 Composition Theorems 67
17.15 Complete Tool Reference 68
17.16 Invariant Verification 68
18 Conclusion 69
A Complete Examples 70
A.1 Example 1: Document Translation (The Glushkov Case) 70
A.2 Example 2: Code Review 71
A.3 Example 3: Data Pipeline 72
A.4 Example 4: Research Task with Error Handling 72
A.5 Example 5: Recursive Decomposition 73
A.6 Example 6: Multi-Stage Transformation with Checkpoints 74
A.7 Example 7: Human-in-the-Loop 75
A.8 Summary of Examples 76
B Anti-Patterns 76
B.1 Anti-Pattern 1: Infinite Reflection 76
B.2 Anti-Pattern 2: Context as Memory 76
B.3 Anti-Pattern 3: Summary of a Summary 76
B.4 Anti-Pattern 4: Top-K Truth 77
B.5 Anti-Pattern 5: Generation Equals Decision 77
B.6 Anti-Pattern 6: Single-Path Confidence 77
B.7 Anti-Pattern 7: Global Memory Pool 77
B.8 Anti-Pattern 8: Unkillable Process 77
B.9 Anti-Pattern 9: Intelligence Metrics 78
B.10 Anti-Pattern 10: Magic Vocabulary 78
B.11 The Final Test 78
C Claude Code NPC Examples 78
C.1 Example 1: Codebase Exploration with Subagent Delegation 78
C.2 Example 2: Code Modification with Permission Handling 79
C.3 Example 3: Hook-Driven Validation Pipeline 80
C.4 Example 4: MCP Integration for External Services 81
C.5 Example 5: Multi-Agent Parallel Processing 81
C.6 Example 6: Plan Mode Research 82

C.7 Example 7: Skill Invocation with Context Forking 83
C.8 Example 8: Complete Development Workflow 84
C.9 Example 9: Error Recovery with Checkpoints 87
Acknowledgments 88
References 88

1 Introduction

In 1964, V.M. Glushkov identified three pillars of computational theory:

(1) **Automata Theory** (Chapter III of *Introduction to Cybernetics* [7]): Formal models of state-based computation
(2) **Self-Organizing Systems** (Chapter IV): Machines that modify their behavior through experience
(3) **Universal Algorithmic Languages** (Chapter V): Formal notations for expressing computation

Traditional programming languages occupy the intersection of (1) and (3): they are formal notations interpreted by deterministic automata with fixed transition functions. The emergence of large language models (LLMs) creates a new possibility: formal notations with deterministic syntax interpreted by self-organizing systems—the intersection of (2) and (3).

This document attempts to formalize one such notation, which we call the **Neural Program Calculus (NPC)**.

1.1 Motivating Example

The following neural program was used to initiate the translation of Glushkov's textbook:

```
execute({
  sample( file.structure );
  for section in file.sections:
    subagent.write_latex(section)
    subagent.validate(completeness, accuracy, polish)
  assemble_final_latex()
  compile_to_pdf()
})
```

Many identifiers in this example are intentionally unresolved at the syntactic level and will be defined via execution context in Section 3. This is neither natural language nor a conventional programming language. It exhibits:

- **Formal structure**: loops, function calls, sequencing
- **Implicit context**: `file` is resolved from conversation, not passed explicitly
- **Intent specification**: `validate(completeness, accuracy, polish)` specifies *what* to check, not *how*
- **Delegation**: `subagent.X` spawns autonomous sub-processes
- **Probabilistic interpretation**: the interpreter determines concrete actions

Our goal is to provide this notation with rigorous foundations.

2 Syntax

2.1 Alphabet

The NPC alphabet Σ consists of the following syntactic token classes:

Definition 2.1 (Terminal Symbols).

$$\Sigma_{\text{keywords}} = \{\texttt{execute}, \texttt{for}, \texttt{in}, \texttt{if}, \texttt{then}, \texttt{else}, \texttt{subagent}\}$$
$$\Sigma_{\text{delimiters}} = \{(,), \{, \}, [,], :, ;, ., ,\}$$
$$\Sigma_{\text{identifiers}} = \{x \mid x \text{ is an alphanumeric string not reserved as a keyword}\}$$
$$\Sigma_{\text{criteria}} = \{\texttt{completeness}, \texttt{accuracy}, \texttt{polish}, \texttt{validity}, \ldots\}$$

The set Σ_{criteria} is extensible and interpreter-defined.

2.2 Grammar

We define the grammar $G = (V, \Sigma, R, S)$ in extended BNF:

Definition 2.2 (NPC Grammar). ⟨*program*⟩ ::= `execute` `(` `{` ⟨*stmt-list*⟩ `}` `)`

⟨*stmt-list*⟩ ::= ⟨*stmt*⟩ (`;` ⟨*stmt*⟩)*

⟨*stmt*⟩ ::= ⟨*action*⟩
| ⟨*loop*⟩
| ⟨*conditional*⟩
| ⟨*delegation*⟩

⟨*action*⟩ ::= ⟨*verb*⟩ `(` ⟨*arg-list*⟩? `)`

⟨*loop*⟩ ::= `for` ⟨*ident*⟩ `in` ⟨*expr*⟩ `:` ⟨*block*⟩

⟨*conditional*⟩ ::= `if` ⟨*predicate*⟩ `:` ⟨*block*⟩ (`else` `:` ⟨*block*⟩)?

⟨*delegation*⟩ ::= `subagent` `.` ⟨*action*⟩

⟨*block*⟩ ::= ⟨*stmt*⟩ | `{` ⟨*stmt-list*⟩ `}`

⟨*verb*⟩ ::= ⟨*ident*⟩ | ⟨*ident*⟩ `_` ⟨*format*⟩

⟨*arg-list*⟩ ::= ⟨*arg*⟩ (`,` ⟨*arg*⟩)*

⟨*arg*⟩ ::= ⟨*expr*⟩ | ⟨*criterion*⟩

⟨*expr*⟩ ::= ⟨*ident*⟩ | ⟨*expr*⟩ `.` ⟨*ident*⟩ | ⟨*literal*⟩

⟨*predicate*⟩ ::= ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*relop*⟩ ⟨*expr*⟩

⟨*criterion*⟩ ::= `completeness` | `accuracy` | `polish` | `validity` | ⟨*ident*⟩

⟨*format*⟩ ::= `latex` | `pdf` | `markdown` | `html` | ⟨*ident*⟩

⟨*literal*⟩ ::= ⟨*string*⟩ | ⟨*number*⟩

Notes on the grammar:

- Newlines are treated as optional semicolons at the concrete syntax level.
- Any identifier may be treated as a criterion, with interpretation delegated to the interpreter.

2.3 Syntactic Sugar

The following abbreviations are permitted:

- Implicit sequencing: newlines may substitute for semicolons
- Method chaining: `x.f().g()` ≡ `let tmp = x.f(); tmp.g()`
- Parallel annotation: `parallel for` explicitly marks parallelizable loops

Concrete Syntax Convention. Throughout this document, code examples employ a Python-like concrete syntax for readability. This surface syntax maps directly to the abstract grammar of Definition 2.2; for instance, `while not task.complete:` corresponds to an iteration construct with negated predicate. All syntactic forms in NPC are deterministic; nondeterminism enters only through semantic interpretation by the neural interpreter defined in Section 3.

3 Semantic Domains

Unlike traditional denotational semantics [23, 33], NPC operates over deterministic syntax but probabilistic, context-dependent semantic interpretation.

3.1 Values

Definition 3.1 (Value Domain). The domain of values $\mathcal{V}$ includes:

$$\begin{aligned}
\mathcal{V} ::= \ & \text{Resource}(r) && \text{(files, documents, URLs)} \\
| \ & \text{Structure}(s) && \text{(parsed representations)} \\
| \ & \text{Collection}(v_1, \ldots, v_n) && \text{(finite ordered sequences of values)} \\
| \ & \text{Artifact}(a) && \text{(generated outputs, possibly typed/serialized)} \\
| \ & \text{Criterion}(c) && \text{(quality predicates)} \\
| \ & \bot && \text{(undefined/failure)}
\end{aligned}$$

Note: The symbol $\bot$ is a meta-symbol from the semantic domain, not a syntactic terminal. It represents computation failure or undefined values.

3.2 Context

Definition 3.2 (Execution Context). An execution context Γ is a tuple:

$$\Gamma = \langle H, E, \rho, \kappa \rangle$$

where:

- H is the conversation history (sequence of prior messages)
- E is the external substrate (files, tools, APIs), modeled abstractly as an external stateful interface
- $\rho : \text{Ident} \rightharpoonup \mathcal{V}$ is the set of syntactic identifiers in $\Sigma_{\text{identifiers}}$
- κ is the capability set (a set of permitted actions / tool accesses)

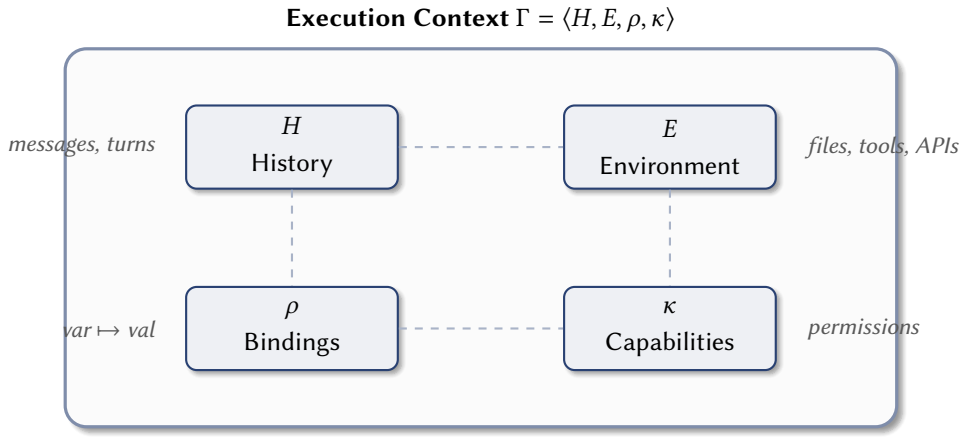

Fig. 1. The four components of an execution context.

We write Program for the set of well-formed NPC programs generated by the grammar in Definition 2.2.

3.3 The Interpretation Function

The key departure from classical semantics: interpretation is performed by a *self-organizing system* $\mathcal{I}$ rather than a deterministic evaluator; we model it as a function that returns a distribution of outcomes.

Definition 3.3 (Neural Interpreter). A neural interpreter is a function:

$$\mathcal{I} : \text{Program} \times \Gamma \to \mathcal{D}(\mathcal{V} \times \Gamma)$$

where $\mathcal{D}(X)$ denotes a (sub-)probability distribution over X.

The interpreter $\mathcal{I}$ is characterized by:

(1) **Training**: $\mathcal{I}$ has been trained on a corpus of (instruction, execution) pairs
(2) **Pattern Recognition**: $\mathcal{I}$ recognizes structural patterns in programs
(3) **Intent Resolution**: $\mathcal{I}$ infers unstated intent from context Γ
(4) **Self-Organization**: $\mathcal{I}$ ...determines concrete semantic actions not specified in the program text

NPC syntax (tokens, grammar, parsing) is deterministic; probabilistic behavior arises only in the mapping from well-formed program text to semantic actions and outcomes via $\mathcal{I}$.

4 Operational Semantics

Definition 4.1 (Operational Judgments). An *operational judgment* is a well-formed assertion describing the execution of a program under an interpreter.

NPC uses a single primary form of operational judgment:

- **Big-step (evaluation) judgment**:

$$\langle P, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu$$

  where $\mu \in \mathcal{D}(\mathcal{V} \times \Gamma)$ is a probability distribution over final values and resulting execution contexts, representing full execution.

All semantic rules in this section define interpreter-relative big-step judgments. NPC does not define an explicit small-step semantics.

All operational rules in this section define interpreter-relative operational judgments.

4.1 Execution

Definition 4.2 (Program Execution).

$$\langle \texttt{execute(\{}S\texttt{\})}, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu \quad \text{iff} \quad \langle S, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu$$

4.2 Sequencing

Definition 4.3 (Sequential Composition). Sequential composition evaluates statements left-to-right, threading the execution context through each step.

$$\langle S_1; S_2, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu$$

where:

(1)
$$\langle S_1, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu_1$$

(2) For each (v, Γ') in the support of μ_1,
$$\langle S_2, \Gamma' \rangle \Downarrow_{\mathcal{I}} \mu_2^{\Gamma'}$$

(3) μ is the distribution obtained by sequencing these evaluations:

$$\mu \;=\; \bigstar_{(v,\Gamma')\sim\mu_1} \mu_2^{\Gamma'}$$

The operator $\bigstar$ denotes distributional sequencing: each possible resulting context from S_1 becomes the starting context for evaluating S_2, and the final distribution aggregates all such execution paths.

4.3 Action Interpretation

This is the crucial rule—actions are interpreted by intent resolution:

Definition 4.4 (Action Semantics). Actions do not have intrinsic operational meaning; instead, they are resolved by the interpreter into executable intent programs.

Let

$$\mathrm{Resolve}_{\mathcal{I}} : \mathrm{Action} \times \Gamma \to \mathrm{Stmt}$$

be an interpreter-dependent resolution function.

If

$$\langle \mathrm{Resolve}_{\mathcal{I}}(a, \Gamma), \Gamma \rangle \Downarrow_{\mathcal{I}} \mu,$$

then

$$\langle a, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu'.$$

The distribution μ' is obtained from μ by binding the final produced value in each resulting context to the distinguished identifier `result`, overwriting any existing binding.

We assume that $\mathrm{Resolve}_{\mathcal{I}}(a, \Gamma)$ always yields a well-formed NPC statement.

4.4 Iteration

Definition 4.5 (Loop Semantics). Iteration evaluates the loop body sequentially over a finite collection, threading the execution context through each iteration.

Let $\rho(coll) = [v_1, \ldots, v_n]$ be a finite collection. Then:

$$\langle \texttt{for } var \texttt{ in } coll : S, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu$$

where:

(1)

$$\langle S[var := v_1], \Gamma \rangle \Downarrow_{\mathcal{I}} \mu_1$$

(2) For each $i = 2, \ldots, n$, and for each (v, Γ_{i-1}) in the support of μ_{i-1},

$$\langle S[var := v_i], \Gamma_{i-1} \rangle \Downarrow_{\mathcal{I}} \mu_i$$

(3) The final distribution is:

$$\mu \;=\; \mu_1 \bigstar \mu_2 \bigstar \cdots \bigstar \mu_n$$

Each iteration executes in the context produced by the previous iteration. Parallel execution is an explicit, separate construct.

Notation: The operator $\bigstar$ denotes distributional sequencing: for each (v, Γ) sampled from μ, evaluation continues from Γ under the next program. $S[var := v]$ denotes the statement S with all free occurrences of variable var replaced by value v.

4.5 Delegation

Definition 4.6 (Subagent Semantics). Delegation spawns an autonomous execution of an action in a derived execution context, then merges the results back into the parent context.

Let a be an action and let Γ_c be a child context derived from Γ by modifying interpreter control parameters to permit autonomous execution.

If

$$\langle a, \Gamma_c \rangle \Downarrow_{\mathcal{I}} \mu_c,$$

then

$$\langle \texttt{subagent}.a, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu_m,$$

where μ_m is obtained by merging each (v, Γ') in the support of μ_c back into the parent context using the context merge operator $\sqcup$.

Definition 4.7 (Context Merge Operator). The context merge operator combines parent and child contexts as follows:

$$\Gamma \sqcup \Gamma' = \langle H \cup H',\ E,\ \rho \oplus \rho',\ \kappa \cap \kappa' \rangle$$

where:

- $H \cup H'$ concatenates conversation histories
- $\rho \oplus \rho'$ merges bindings, with child bindings taking precedence
- $\kappa \cap \kappa'$ restricts capabilities to their intersection
- E is assumed shared and immutable

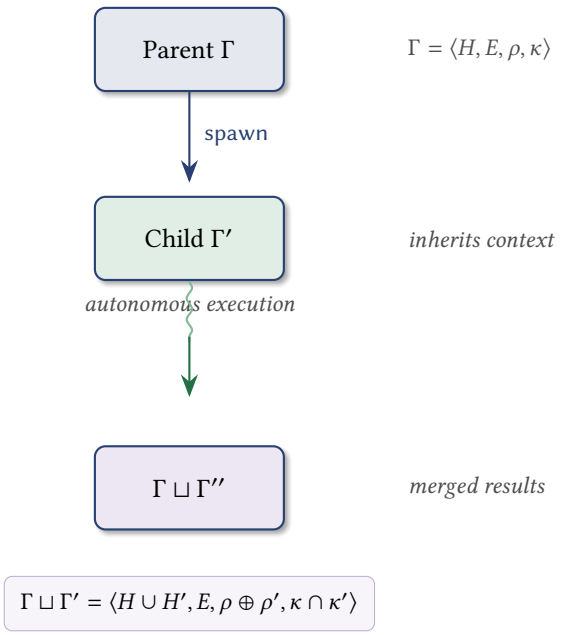

Fig. 2. Subagent delegation: parent spawns child with inherited context; results merged back.

5 Validation Semantics

Validation in NPC is declarative: criteria specify *what* properties should hold of an artifact, not *how* those properties are checked. Validation does not introduce new computation; it constrains the acceptance of previously produced results.

Definition 5.1 (Validation). Let $C = \{c_1, \ldots, c_k\}$ be a finite set of validation criteria. The validation construct is defined as:

$$\texttt{validate}(C) \;\equiv\; \bigwedge_{i=1}^{k} \text{Check}_{\mathcal{I}}(c_i, \Gamma.\texttt{result})$$

where:

- $\Gamma.\texttt{result}$ is the value most recently produced by execution
- $\text{Check}_{\mathcal{I}}(c, v)$ is a boolean judgment returned by the interpreter $\mathcal{I}$, indicating whether value v satisfies criterion c

Validation does not modify the execution context Γ except through interpreter-internal state (e.g., confidence estimates, implicit judgments). If validation fails, subsequent control flow (e.g., retries, counteragents, or aborts) must be specified explicitly by the program.

Definition 5.2 (Standard Criteria). The following criteria are commonly used but are not exhaustive:

`completeness` :	All required elements of the input are reflected in the output
`accuracy` :	The output preserves the intended semantics of the input
`polish` :	The output meets stylistic and presentation standards
`validity` :	The output is well-formed with respect to its target format

Criteria are interpreted relative to:

- the execution context Γ
- the interpreter's training and internal representations
- implicit task norms inferred during execution

Non-Guarantee. Successful validation does *not* imply objective correctness. It asserts only that the interpreter $\mathcal{I}$ judges the result to satisfy the stated criteria under the current context.

Failure Signaling. Validation failure is observable only through program structure (e.g., conditional branches, counteragents, or confidence checks); the calculus itself does not enforce automatic retries or halting.

6 Type System

NPC admits a *soft* type system intended to guide interpretation rather than enforce correctness. Types constrain expectations about format and compatibility, but they do not determine execution behavior.

The NPC type system is:

- **Advisory**, not enforced
- **Non-sound**, relative to interpreter behavior
- **Format-oriented**, not semantic

6.1 Format Types

Definition 6.1 (Format Types). The set of NPC format types is defined inductively as:

$$\tau ::= \texttt{text} \mid \texttt{markdown} \mid \texttt{latex} \mid \texttt{pdf} \mid \texttt{html} \mid \texttt{code}[\ell] \mid \tau \times \tau \mid [\tau]$$

where ℓ denotes a programming language identifier (e.g., `python`, `javascript`, `rust`).

Format types describe *representation constraints*, not semantic meaning. Two values of the same format type may differ arbitrarily in correctness, completeness, or intent satisfaction.

6.2 Action Signatures

Definition 6.2 (Action Signatures). Actions may be annotated with implicit type signatures of the form:

$$a : \tau_1 \longrightarrow \tau_2$$

indicating the expected transformation of representation formats.

Common examples include:

$$\begin{aligned} \texttt{sample} &: \texttt{Resource} \longrightarrow \texttt{Structure} \\ \texttt{write_latex} &: \texttt{text} \longrightarrow \texttt{latex} \\ \texttt{compile_latex_to_pdf} &: \texttt{latex} \longrightarrow \texttt{pdf} \\ \texttt{validate} &: \tau \times \texttt{Criterion}^* \longrightarrow \texttt{bool} \end{aligned}$$

These signatures are *heuristic contracts*. The interpreter $\mathcal{I}$ may violate them, reinterpret them, or partially satisfy them depending on context, training, and capability constraints.

6.3 Well-Typedness

Definition 6.3 (Well-Typed Program). An NPC program is *well-typed* if, under reasonable interpreter assumptions:

(1) Format transformations are compatible (e.g., no implicit `text` $\longrightarrow$ `pdf` without an intermediate step)
(2) Loop variables preserve a consistent expected format across iterations
(3) Validation criteria are applicable to the preceding result's format

Well-typedness is neither decidable nor enforceable within NPC itself. It is an *a priori design property*, not a runtime guarantee.

Advisory Nature. The interpreter may attempt to execute ill-typed programs. Type violations increase the probability of failure but do not prevent execution.

Failure Modes. Type mismatches typically manifest as:

- Interpreter confusion
- Hallucinated intermediate formats
- Silent coercions
- Validation failure

Design Intent. The type system exists to:

- Communicate programmer intent
- Enable human reasoning about programs
- Support validation and counteragent design

It does not—and cannot—provide safety or correctness guarantees.

7 Design Patterns as Compositional Operators

This section formalizes empirically observed *design patterns* for neural program execution as compositional operators. These operators do not transform program syntax or values directly; instead, they transform *execution regimes* under which programs are interpreted.

Accordingly, the operators defined here are:

- **Meta-operational**, acting on execution structure
- **Interpreter-relative**, not intrinsic program semantics
- **Non-transparent**, in that they may introduce additional computation not explicit in the program text

Their purpose is to manage bounded context, probabilistic failure, and coordination across multiple executions.

7.1 The Continuation Operator

The finite context window of neural interpreters necessitates explicit continuation mechanisms.

Definition 7.1 (Continuation Operator). The **continuation operator** $\mathcal{K}$ maps a program P to a wrapped program $\mathcal{K}(P)$ that enforces resumable execution:

$$\mathcal{K}(P) \equiv \texttt{while not } P.\texttt{done}: \{\texttt{load_checkpoint}(); P.\texttt{continue}(); \texttt{save_checkpoint}()\}$$

The continuation operator does not preserve single-run semantics; it preserves *eventual task intent* under bounded context.

Constraint. $\mathcal{K}$ assumes:

- Checkpointable intermediate state
- An externally observable notion of completion

7.2 The Compression Operator

Context is a finite resource and must be actively managed.

Definition 7.2 (Compression Operator). The **compression operator** $\downarrow$ maps an execution context to a reduced context:

$$\downarrow : \Gamma \to \Gamma' \quad \text{such that} \quad |\Gamma'| < |\Gamma|$$

while approximately preserving task-relevant information.

Compression is explicitly lossy and must be treated as such.

Common strategies include:

- **Summarization**: replacing history with a semantic summary
- **Windowing**: retaining only recent context
- **Selective filtering**: retaining high-relevance elements

Theorem 7.3 (Compression–Fidelity Tradeoff). *For any compression operator $\downarrow$ with ratio $r = |\Gamma'|/|\Gamma|$:*

$$\mathbb{E}\left[\|Rel_T(\Gamma') - Rel_T(\Gamma)\|\right] \geq f(r)$$

for a monotonically decreasing function f. This result is a domain-specific instance of rate-distortion theory [25].

Compression does not commute with validation or counteragent application.

7.3 The Journaling Operator

Persistent logging is required for recovery, debugging, and auditability.

Definition 7.4 (Journaling Operator). The **journaling operator** $\mathcal{J}$ augments execution with append-only trace logging:

$$\mathcal{J}(P) \equiv P \parallel \texttt{journal.append}(\text{trace}(P))$$

Journaling introduces side effects but does not alter the intended task outcome.

Axiom 7.1 (Journal Completeness). *If the journal captures all state transitions relevant to execution, then replay is possible up to interpreter nondeterminism.*

7.4 The Semantic Index Operator

Large corpora require explicit retrieval structure.

Definition 7.5 (Semantic Index Operator). The **semantic index operator** $\mathcal{X}$ constructs a retrieval index:

$$\mathcal{X}(\text{corpus}) = \{(k_i, v_i) \mid k_i = \text{embed}(v_i),\ v_i \in \text{corpus}\}$$

The index is advisory: similarity does not imply relevance.

7.5 The Counteragent Operator

Adversarial checking improves reliability through independent critique.

Definition 7.6 (Counteragent Operator). The **counteragent operator** $\mathcal{A}$ transforms a program P into a two-stage execution:

$$\mathcal{A}(P) \equiv P;\ \texttt{spawn adversary} : \texttt{find_flaws}(P.\texttt{output});\ \texttt{if flaws: revise}()$$

The counteragent is not guaranteed to be correct; it is guaranteed only to be *independent*.

Conjecture 7.1 (Counteragent Improvement). *For tasks with externally verifiable correctness:*

$$\Pr[\mathit{correct}(\mathcal{A}(P).\mathit{output})] > \Pr[\mathit{correct}(P.\mathit{output})]$$

7.6 The Ledger Operator

Auditability requires immutable history.

Definition 7.7 (Ledger Operator). The **ledger operator** $\mathcal{L}$ strengthens journaling with content-addressed immutability:

$$\mathcal{L}(P) \equiv \mathcal{J}(P) \ \text{ with } \ \text{Entry.hash} = H(\text{Entry} \| \text{prev_hash})$$

7.7 Operator Composition

These operators compose at the *execution level*, not the value level.

Definition 7.8 (Robust Execution Strategy). A robust execution strategy is an operator composition:

$$\text{Robust}(P) = \mathcal{K}(\mathcal{J}(\mathcal{A}(P)))$$

Proposition 7.9 (Associativity). *Operator composition is associative up to execution equivalence:*

$$(\mathcal{K} \circ \mathcal{J}) \circ \mathcal{A} \simeq \mathcal{K} \circ (\mathcal{J} \circ \mathcal{A})$$

but not commutative.

8 Connection to Glushkov's Theory

This section situates the Neural Program Calculus (NPC) within the framework developed by V.M. Glushkov in *Introduction to Cybernetics* (1964). The connection is structural rather than literal: NPC preserves Glushkov's notion of deterministic symbolic structure while generalizing the nature of the interpreting system.

8.1 Alphabetic Operators Revisited

Glushkov defined *alphabetic operators* as deterministic transformations over strings drawn from a finite alphabet Σ:

$$F : \Sigma^* \to \Sigma^*$$

These operators describe symbolic manipulation independent of the physical implementation of the machine executing them.

NPC preserves this symbolic determinism at the syntactic level.

Deterministic Alphabet. In NPC, the alphabet Σ and the grammar generating Σ^* are fully deterministic. Tokens, parsing, and well-formedness are classical and admit no probabilistic interpretation.

Generalization Point. The generalization occurs *only* at the interpretation layer: NPC programs are not executed by a fixed transition function, but by a learned, self-organizing interpreter.

Definition 8.1 (Interpreter-Relative Alphabetic Operator). An NPC program P induces an interpreter-relative transformation:

$$[\![P]\!]_I : \Sigma^* \times \Gamma \ \to \ \mathcal{D}(\Sigma^* \times \Gamma)$$

mapping a symbolic program string and execution context to a distribution over resulting symbolic artifacts and contexts.

Thus, NPC does *not* replace $F : \Sigma^* \to \Sigma^*$ with a stochastic alphabet. It replaces the *executor* of F.

8.2 Neural Automata

Classical Glushkov automata consist of:

$$(Q, \Sigma, \delta, q_0, F) \quad \text{with} \quad \delta : Q \times \Sigma \to Q$$

NPC generalizes the transition function while preserving the symbolic interface.

Definition 8.2 (Neural Automaton). A **neural automaton** is a tuple

$$\mathcal{N} = (Q, \Sigma, \delta_I, q_0, F)$$

where:

- Q is a (possibly continuous) state space
- Σ is a deterministic input alphabet
- $\delta_I : Q \times \Sigma \to \mathcal{D}(Q)$ is an interpreter-computed transition distribution
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the acceptance condition

The probabilism lies in δ_I, not in Σ or Q.

Proposition 8.3 (Embedding of Classical Automata). *Every deterministic finite automaton embeds into a neural automaton via:*

$$\delta_I(q, a) = \mathrm{Dirac}(\delta(q, a))$$

8.3 Neural Languages

Classical automata define languages as sets of accepted strings. Neural automata define *acceptance likelihoods*.

Definition 8.4 (Neural Language). Given $\epsilon \in (0, 1)$, the neural language recognized by $\mathcal{N}$ is:

$$L_\epsilon(\mathcal{N}) = \{\, w \in \Sigma^* \mid \Pr[\delta_{\mathcal{I}}^*(q_0, w) \in F] \geq 1 - \epsilon \,\}$$

This definition preserves the notion of language while relaxing acceptance from a binary predicate to a probabilistic threshold.

8.4 Self-Organization as Context Dependence

Glushkov characterized self-organizing systems as those whose behavior changes in response to experience.

In NPC, this manifests as context-dependent interpretation.

Proposition 8.5 (In-Context Self-Organization). *For an action a and successive contexts Γ_t, Γ_{t+1}:*

$$\mathcal{I}(a, \Gamma_t) \neq \mathcal{I}(a, \Gamma_{t+1})$$

in general, even when a is syntactically identical.

The symbolic program remains unchanged; the effective transition function evolves through context accumulation.

8.5 Glushkov's Synthesis Realized

NPC realizes Glushkov's anticipated synthesis:

(1) **Automata Theory**: deterministic symbolic structure
(2) **Self-Organization**: interpreter behavior adapts via context
(3) **Universal Languages**: programs specify transformations

The novelty is architectural, not symbolic: the executor is no longer a fixed automaton but a bounded, learned, probabilistic system.

NPC therefore extends Glushkov's framework without violating its core distinction between symbolic form and machine realization.

9 Probabilistic Correctness and Verification

Classical program verification establishes deterministic properties of programs. Neural program verification must instead reason about *probability distributions over outcomes* produced by interpreter-mediated execution.

Correctness in NPC is therefore not a binary predicate but a quantitative, context-dependent property.

9.1 Partial Correctness

Definition 9.1 (Probabilistic Partial Correctness). Let P be a neural program, ϕ a precondition on execution contexts, and ψ a postcondition.

P is (ϵ, δ)-**partially correct** with respect to (ϕ, ψ) if:

$$\Pr_{\Gamma \sim \mu_0}[\Gamma \models \phi] \geq 1 - \epsilon \quad \Rightarrow \quad \Pr_{(v, \Gamma') \sim \mu}[\Gamma' \models \psi] \geq 1 - \delta$$

where:

$$\langle P, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu$$

Partial correctness does not imply termination, nor does it imply that ψ holds in any individual execution.

Definition 9.2 (Correctness Measure). Given a program P and postcondition ψ, define:

$$\text{Correct}(P, \psi) = \Pr_{(v,\Gamma')\sim\mu} \left[\Gamma' \models \psi\right]$$

This quantity is interpreter- and context-relative.

9.2 Verification Strategies

Because no single execution is authoritative, verification relies on meta-strategies that reduce failure probability.

(1) **Counteragent Verification**
Apply the counteragent operator $\mathcal{A}$:

$$P' = \mathcal{A}(P)$$

and measure the observed failure rate over repeated executions.

(2) **Redundant Execution**
Execute P multiple times under independently perturbed contexts:

$$\text{Vote}_n(P) = \text{mode}\{\mathcal{I}(P, \Gamma_i) : i = 1, \ldots, n\}$$

Independence is heuristic, not guaranteed. Note: this is a repetition code; correlated failures reduce effective redundancy. Counteragent verification (Section 7.5) provides diversity coding, which is more robust to systematic errors.

(3) **Explicit Checkpoints**
Insert validation and assertion points:

$$P' = P_1;\ \texttt{assert}(\phi_1);\ P_2;\ \texttt{assert}(\phi_2);\ \ldots$$

(4) **Human-in-the-Loop**
Require explicit human approval at designated decision points.

These strategies reduce risk but do not eliminate failure.

9.3 Compositional Verification

Probabilistic correctness composes approximately.

Theorem 9.3 (Sequential Composition). *If P_1 is $(0, \delta_1)$-correct for (ϕ, ψ) and P_2 is $(0, \delta_2)$-correct for (ψ, χ), then the sequential composition $P_1; P_2$ satisfies:*

$$\text{Correct}(P_1; P_2, \chi) \ \geq\ 1 - (\delta_1 + \delta_2)$$

Proof. Let F_1 denote the event that S_1 fails (i.e., produces $\perp$ or violates its specification) and F_2 denote the corresponding event for S_2.

We require successful completion of both stages. The failure event for the composition is $F_1 \cup F_2$.

By assumption, $\Pr[F_1] \leq \delta_1$ and $\Pr[F_2 \mid \neg F_1] \leq \delta_2$.

Applying the union bound:

$$\Pr[F_1 \cup F_2] \leq \Pr[F_1] + \Pr[F_2] \leq \delta_1 + \delta_2$$

Therefore the probability of successful completion is:

$$\Pr[\neg(F_1 \cup F_2)] \geq 1 - (\delta_1 + \delta_2)$$

□

The tighter bound $1 - (\delta_1 + \delta_2 - \delta_1\delta_2)$ holds only under independence assumptions, which are generally unjustified for neural interpreters.

Theorem 9.4 (Parallel Composition). *If P_1 and P_2 are executed in parallel under approximately independent contexts, then:*

$$\text{Correct}(P_1 \parallel P_2) \approx \text{Correct}(P_1) \cdot \text{Correct}(P_2)$$

Parallel independence must be argued, not assumed.

9.4 The Halting Problem

Theorem 9.5 (Neural Halting Undecidability). *It is undecidable whether an arbitrary neural program halts.*

Proof. Neural programs can encode instructions of the form "simulate a Turing machine with transition table T on input w." Given a sufficiently expressive interpreter, execution simulates the machine's behavior.

Since the halting problem for Turing machines is undecidable [12, 27], so is halting for neural programs. (This reduction assumes an interpreter capable of faithful Universal Turing Machine simulation; for bounded practical interpreters, the problem may be decidable but intractable.) □

Definition 9.6 (Manifestly Terminating). A neural program is **manifestly terminating** if termination follows from explicit syntactic bounds, such as:

- Finite iteration over bounded collections
- Bounded loops with explicit limits
- Timeouts enforced by the execution harness

This class corresponds to programs expressible with bounded recursion; cf. primitive recursive functions (Péter, 1934).

Proposition 9.7. *Halting is decidable for manifestly terminating neural programs.*

9.5 Confidence Estimation

Neural interpreters may provide self-assessed confidence estimates.

Definition 9.8 (Self-Assessed Confidence). For output o produced during execution, define:

$$\text{conf}(o) = \mathcal{I}(\texttt{"How confident are you in this output?"}, \Gamma \cup \{o\})$$

Confidence estimates are additional observations, not guarantees.

Conjecture 9.1 (Calibration). *For well-trained interpreters,*

$$\Pr[\text{correct}(o) \mid \text{conf}(o) = c] \approx c$$

Calibration is empirical and task-dependent.

Confidence gating enables risk-aware execution but must not be treated as a proof of correctness.

10 Axioms and Theorems

This section states the foundational assumptions and derived properties underlying the Neural Program Calculus. These statements are scoped to interpreters $\mathcal{I}$ intended to execute NPC programs as specified in this document.

10.1 Axioms

Axioms assert structural properties required for NPC to function as a usable calculus. They are not empirical claims about all language models.

Axiom 10.1 (Context Inheritance). *Let*

$$\langle \texttt{subagent}.a, \Gamma \rangle \Downarrow_{\mathcal{I}} \mu.$$

Then for all (v, Γ') *in the support of* μ*,*

$$H_{\Gamma'} \supseteq H_{\Gamma}.$$

That is, subagent execution may extend but does not erase prior history.

Axiom 10.2 (Intent Monotonicity). *If program* P *specifies intent* I *and the interpreter resolves this to* I'*, then:*

$$I \subseteq I'.$$

The interpreter may elaborate or over-satisfy intent, but it must not silently omit specified intent.

Axiom 10.3 (Explicit Failure). *If an action cannot be resolved within the interpreter's capability set, execution must yield* $\perp$ *or trigger an explicit failure signal.*

10.2 Propositions

Propositions are logical consequences of the axioms and previously defined semantics.

Proposition 10.1 (Delegation Isolation). *Subagent execution cannot reduce the parent capability set:*

$$\kappa_{\Gamma'} \subseteq \kappa_{\Gamma}$$

for all merged contexts Γ' *produced by subagent delegation.*

Proposition 10.2 (Validation Soundness Relative to Interpreter). *If*

$$\texttt{validate}(C) = \texttt{true},$$

then for all $c \in C$*,*

$$\mathit{Check}_{\mathcal{I}}(c, \texttt{result}) = \texttt{true}.$$

This assertion holds only relative to the interpreter's internal criteria.

10.3 Conjectures

Conjectures describe empirically observed regularities across practical interpreters. They are falsifiable and model-dependent.

Conjecture 10.1 (Task Decomposition). *If a task* T *admits decomposition*

$$T = \bigoplus_{i=1}^{n} T_i$$

with approximately independent subtasks, then:

$$\texttt{execute}(T) \;\simeq\; \texttt{for } t \texttt{ in } [T_1, \ldots, T_n] : \texttt{execute}(t); \texttt{assemble}()$$

where $\simeq$ *denotes equivalence up to execution distribution.*

Conjecture 10.2 (Parallel Equivalence). *For independent iterations,*

$$[\![\texttt{for } x \texttt{ in } L : S]\!]_{seq} \;\simeq\; [\![\texttt{for } x \texttt{ in } L : S]\!]_{\|}$$

provided that execution contexts do not interfere.

Conjecture 10.3 (Counteragent Improvement—Aggregated). *Conjecture 7.1 strengthens under aggregation: for tasks with externally checkable correctness,*

$$\Pr[\mathit{correct}(\mathcal{A}(P).\mathit{output})] > \Pr[\mathit{correct}(P.\mathit{output})]$$

under repeated execution and aggregation.

10.4 Scope Note

None of the conjectures above are asserted to hold for all interpreters. They describe regularities observed in:

- Large language models trained with instruction-following objectives
- Systems employing explicit counteragents, validation, and retries
- Harnessed execution environments enforcing boundedness

Violations of these conjectures indicate either:

- An interpreter outside the intended NPC class, or
- A failure of the underlying design assumptions

11 Extensions

This section outlines potential extensions to the Neural Program Calculus. These extensions expand expressive convenience but do not alter the core semantic model established in Sections 3–10.

All extensions described here satisfy the following constraints:

- Syntax remains deterministic
- Semantics remain interpreter-relative
- No extension introduces implicit guarantees

Accordingly, these constructs should be understood as *desugaring targets*: they may be translated into core NPC constructs prior to interpretation.

11.1 Structured Error Handling

Neural execution fails probabilistically and often partially. Explicit error-handling constructs provide structure for such failure without implying recoverability guarantees.

⟨*stmt*⟩ ::= ...
| 'try' ':' ⟨*block*⟩ 'on_failure' ':' ⟨*block*⟩

Intended Semantics. The `try/on_failure` construct does not intercept low-level interpreter failures. Instead, it provides a *semantic framing* that permits the interpreter to:

- Surface uncertainty
- Abandon partial results
- Select fallback strategies

Desugaring. A `try/on_failure` block may be desugared into explicit conditional control flow over interpreter-produced failure signals.

Non-Guarantee. Failure handling does not imply correctness, completeness, or graceful recovery. It merely prevents silent continuation after unresolved failure.

11.2 Quantitative Validation Criteria

Validation criteria may be extended with quantitative thresholds.

⟨*criterion*⟩ ::= ...
| ⟨*ident*⟩ ⟨*relop*⟩ ⟨*number*⟩

Example. `validate(accuracy > 0.95)`

Interpretation. Quantitative criteria are *ordinal signals*, not calibrated measurements. The interpreter determines:

- The scale on which the criterion is evaluated
- The meaning of comparison operators
- The reliability of the estimate

Constraint. Programs must not assume numeric stability or cross-run comparability of quantitative scores.

11.3 Named Definitions and Abstraction

Reusable abstractions improve readability and modularity but do not introduce true function semantics.

⟨*stmt*⟩ ::= ...
  | 'define' ⟨*ident*⟩ '(' ⟨*params*⟩? ')' ':=' ⟨*block*⟩

Example.
```
define process_section(s) := {
  subagent.write_latex(s)
  subagent.validate(completeness, accuracy)
}
```

Semantics. Definitions bind a syntactic identifier to a block of statements. Invocation is resolved via interpreter-relative substitution, not lexical closure.

Limitation. Definitions do not guarantee:

- Referential transparency
- Termination
- Argument immutability

They are organizational devices, not functions in the lambda-calculus sense.

11.4 Streaming and Incremental Execution

NPC may express streaming transformations over large or unbounded collections.

⟨*stmt*⟩ ::= ...
  | 'stream' ⟨*expr*⟩ '|>' ⟨*verb*⟩

Example. `stream file.lines |> translate_to_english`

Operational Interpretation. Streaming permits the interpreter to:

- Process elements incrementally
- Interleave computation with I/O
- Apply backpressure or truncation

Constraint. Streaming does not imply:

- Deterministic ordering
- Complete consumption of the stream
- Idempotent transformation

Termination and completeness must be enforced via external bounds.

11.5 Status of Extensions

All extensions in this section are:

- **Optional**: interpreters may ignore unsupported constructs
- **Conservative**: no extension strengthens correctness claims
- **Non-fundamental**: all can be eliminated via desugaring

They exist to improve human expressivity, not to expand the theoretical power of the calculus.

12 Discussion

12.1 Related Work and Intellectual History

The Neural Program Calculus builds upon and is substantially inspired by prior work in the emerging field of LLM-oriented programming languages.

12.1.1 SudoLang. The most direct intellectual predecessor is **SudoLang**, created by Eric Elliott and the Parallel Drive team in 2023 [6]. SudoLang pioneered several concepts that NPC formalizes:

- **Natural language constraint programming**: SudoLang demonstrated that LLMs can reliably interpret hybrid natural-pseudocode specifications, treating constraints as program logic.
- **Referential omnipotence**: SudoLang introduced the principle that functions need not be explicitly defined—the interpreter infers their behavior from naming and context. NPC formalizes this as the Resolve_I function.
- **The |> pipe operator**: SudoLang adopted functional composition via pipes, which NPC inherits for streaming operations.
- **Interface-based structure**: SudoLang's typed interfaces for modular program components influenced NPC's approach to format types.
- **/commands**: SudoLang's command syntax for defining interaction interfaces is echoed in NPC's action semantics.

12.1.2 Key Differences from SudoLang. While NPC owes a substantial debt to SudoLang, it diverges in several important respects:

Aspect	SudoLang	NPC
Primary goal	Practical programming language for LLM collaboration	Formal calculus with mathematical semantics
Semantics	Informal, demonstrated by examples	Operational semantics with inference rules
Theoretical grounding	Empirical ("it works")	Connected to automata theory, Glushkov's framework
Probabilistic treatment	Implicit	Explicit ($\mathcal{D}(\cdot)$ distributions, (ϵ, δ)-correctness)
Compositional operators	Ad hoc patterns	Formalized ($\mathcal{K}, \downarrow, \mathcal{J}, \mathcal{A}$, etc.)
Verification	Not addressed	Probabilistic correctness framework
Target audience	Practitioners	Theoreticians and language designers

In essence, SudoLang is to NPC as Lisp was to the lambda calculus: SudoLang is a practical language that demonstrated feasibility; NPC attempts to provide the underlying formal theory.

12.1.3 Other Related Work. Several other efforts have explored LLM-oriented programming:

- **Prompting frameworks** (LangChain [4], DSPy [16]): Focus on composing LLM calls but lack formal semantics.
- **Natural language programming** research: Academic work on parsing natural language to code, typically targeting compilation to traditional languages rather than direct LLM interpretation.
- **Chain-of-thought prompting** [17, 32]: Demonstrated that structured reasoning improves LLM performance, supporting the value of formal program structure. Tree-of-thought [35] and self-consistency [31] extend this to more complex reasoning patterns.
- **Constitutional AI** [1]: Anthropic's work on constraint-based AI behavior, resonant with SudoLang's constraint philosophy.
- **Agentic systems** [19, 34]: ReAct and generative agents demonstrate LLM-based autonomous action, which NPC seeks to formalize.

12.1.4 Theoretical Foundations. NPC draws on several foundational areas:

- **Denotational semantics** [21, 23, 33]: Classical frameworks for defining programming language meaning. NPC adapts these to probabilistic interpretation.
- **Type theory** [20]: NPC's "soft types" relate to gradual typing [26], where type annotations provide partial guarantees.
- **Probabilistic programming** [2, 3, 8]: Languages like Church, Stan, and Pyro formalize probabilistic inference. NPC's probabilistic semantics share this spirit but differ in mechanism: NPC's distributions arise from neural interpretation, not explicit probability models.
- **Program synthesis** [9, 28]: Automatic program generation from specifications. NPC inverts this: rather than synthesizing deterministic programs from specs, it expresses intent for probabilistic execution.
- **Information theory** [24, 25]: Compression bounds and rate-distortion theory inform NPC's analysis of semantic loss under context compression.
- **Neural network theory** [5, 13, 18, 22]: Universal approximation theorems establish that neural networks can approximate arbitrary functions, underpinning the generality of neural interpretation. The deep learning revolution [10] enabled practical realization of these theoretical results.
- **Scaling laws** [11, 15]: Empirical relationships between model size, data, and capability inform understanding of interpreter behavior.
- **LLM reliability** [14]: Survey work on hallucination and reliability failures motivates NPC's emphasis on validation and bounded trust.

12.1.5 Historical Context. The intellectual lineage can be traced:

(1) **1945**: von Neumann's report on the EDVAC [30] establishes the stored-program architecture—computation that can modify its own instructions.
(2) **1964**: Glushkov's *Introduction to Cybernetics* establishes automata theory, self-organizing systems, and universal algorithmic languages as the three pillars.
(3) **2017**: The Transformer architecture [29] enables practical large language models.
(4) **2022**: ChatGPT demonstrates that LLMs can reliably follow complex, structured instructions.
(5) **2023**: SudoLang formalizes the practice of LLM-oriented programming into a language specification.
(6) **2026**: NPC attempts to provide mathematical foundations for this emerging paradigm, connecting back to Glushkov's framework.

The recursion is notable: Glushkov theorized self-organizing systems that would interpret algorithmic languages; sixty years later, such systems exist and require new formal frameworks—which themselves draw on Glushkov's foundations.

12.2 Comparison with Traditional Formalisms

Property	ALGOL-60	Lambda Calculus	NPC
Deterministic	Yes	Yes	No[†]
Explicit state	Yes	No	Partial
Intent resolution	No	No	Yes[‡]
Self-organizing interpreter	No	No	Yes[§]
Formal grammar	Yes	Yes	Yes
Decidable type system	Yes	Yes	No[‖]

Notes on comparability:

[†] NPC syntax is deterministic; nondeterminism arises from interpreter-mediated semantic resolution.

[‡] "Intent resolution" is a novel property without direct analog in traditional formalisms—the comparison cell for ALGOL/Lambda Calculus indicates absence, not failure.

[§] "Self-organizing" in Glushkov's sense: the interpreter modifies its behavior based on experience (training). Traditional interpreters are fixed functions.

[‖] NPC types are advisory annotations, not enforced constraints. This is fundamentally different from ALGOL's static typing or Lambda Calculus's type systems. A more precise characterization: NPC has a *format annotation system*, not a type system in the traditional sense.

12.3 Open Problems

(1) **Decidability**: Is well-typedness decidable for NPC?
(2) **Equivalence**: When are two NPC programs equivalent?
(3) **Completeness**: What class of tasks is expressible in NPC?
(4) **Verification**: Can we verify that an interpreter correctly implements NPC semantics?
(5) **Composition**: What algebraic laws govern program composition?

13 Case Study: Production Counteragent System

We present a detailed analysis of a production neural program implementing the counteragent operator $\mathcal{A}$. This system, deployed in a commercial sales automation platform, demonstrates several advanced techniques for controlling LLM behavior through semantic algorithm specification.

13.1 System Overview

The `CounterAgent` daemon monitors a queue of proposed agent actions and makes approve/reject/abort decisions by comparing each action against patterns learned from human-curated examples. This is a direct instantiation of the counteragent operator:

$$\mathcal{A}(\text{Annie}) = \text{Annie}; \texttt{CounterAgent.evaluate()}; \texttt{if rejected: revise()}$$

where "Annie" is the primary agent proposing customer outreach actions.

13.2 The Neural Program

The core of the system is a neural program embedded in the LLM system prompt. We present it with annotations:

```
try {
  # STEP 1: Load and analyze the example actions
  examples = load_provided_examples()
  approved_examples = filter(examples, status='approved')
  rejected_examples = filter(examples, status='rejected')
  aborted_examples = filter(examples, status='aborted')

  # STEP 2: Extract patterns from approval_notes
  for example in examples:
    extract_pattern_from_approval_notes(example)
    identify_common_phrases_in_feedback(example)
    detect_reviewer_preferences(example)

  # STEP 3: Check prompt instruction compliance
  if current_case.has_prompt_instructions():
    compliance_check = verify_prompt_compliance(current_case)
    if not compliance_check.is_compliant():
      return {
        status: "rejected",
        notes: "Doesn't follow prompt instructions: "
               + compliance_check.get_violations(),
        confidence_score: "95"
      }

  # STEP 4: Check dashboard for missed information
  dashboard_analysis = analyze_dashboard_data(current_case.dashboard_data
      )
  if dashboard_analysis.has_critical_info_annie_missed():
    return {
      status: "rejected",
      notes: "Dashboard shows: " + dashboard_analysis.get_missed_info(),
      confidence_score: calculate_confidence()
    }

  # STEP 5: Pattern matching against approved examples
  approved_patterns = extract_common_patterns(approved_examples)
  if current_action_matches_approved_patterns(current_case,
      approved_patterns):
    similarity_score = calculate_similarity_to_approved()
    if similarity_score > 0.8:
      return {
        status: "approved",
        notes: "Matches approved pattern: " + best_matching_pattern,
        confidence_score: str(int(similarity_score * 100))
      }

  # STEP 6: Pattern matching against rejected examples
  rejected_patterns = extract_common_patterns(rejected_examples)
  if current_action_matches_rejected_patterns(current_case,
      rejected_patterns):
    return {
```

```
      status: "rejected",
      notes: "Similar to rejected pattern: " + find_rejection_reason(),
      confidence_score: calculate_confidence()
    }

  # STEP 7: Check for repetitive rejection patterns
  if detect_repetitive_rejection_pattern(current_case, recent_rejections)
      :
    if should_abort_entire_task():
      return {
        status: "aborted",
        notes: "ABORT_TASK: " + get_pattern_reason(),
        confidence_score: "98"
      }

  # STEP 8: Final decision
  return { status: string; notes: string; confidence_score: "1-100"; }
}
catch (e: any) {
  // Graceful degradation
}
```

Listing 1. Production Counteragent Neural Program (Simplified)

13.3 Novel Techniques

This production system introduces several techniques not previously formalized:

13.3.1 Technique 1: Semantic Try-Catch Wrapper.

Definition 13.1 (Semantic Exception Handling). Wrapping a neural program in `try { ... } catch(e) { ... }` provides the interpreter with an error-handling mental model:

$$\texttt{try}(P) \equiv P \cup \{\texttt{if undefined}(P.\text{output}) : \texttt{throw exception}\}$$

This gives the LLM explicit permission to signal confusion rather than hallucinate. The production system parses for “‘`exception` blocks as an alternative to “‘`final-output`.

13.3.2 Technique 2: Code-Fenced Output Typing.

Definition 13.2 (Fenced Output Block). A **fenced output block** constrains LLM output to a parseable region:

````
```final-output
{ status: "approved" | "rejected" | "aborted";
  notes: string;
  confidence_score: "1-100"; }
```
````

The TypeScript wrapper extracts content between fences:

```
const parts = response.split('```final-output');
const finalOutput = parts[1].split('```')[0].trim();
const result = JSON.parse(finalOutput);
```

This technique provides:

- **Encoding control**: Output appears in predictable location
- **Type specification**: The schema inside the fence guides structure
- **Separation of reasoning from output**: Chain-of-thought can precede the fence

13.3.3 Technique 3: Numbered Step Forcing.

Definition 13.3 (Step-Forced Execution). Explicit numbered steps (`# STEP 1:`, `# STEP 2:`, ...) create cognitive checkpoints:

$$P = S_1; S_2; \ldots; S_n \quad \text{with explicit labels}$$

Combined with the execution instruction:

> *"Execute your program on this input. First, start with outlining your assumptions, and then run the program by reasoning through each step of it."*

This forces the LLM to:

(1) State assumptions explicitly (debuggable)
(2) Process each step sequentially (auditable)
(3) Show work before conclusion (chain-of-thought)

13.3.4 Technique 4: Meta-Signal Protocol.

Definition 13.4 (Meta-Signal). A **meta-signal** is a string pattern in LLM output that triggers programmatic behavior beyond the immediate response:

$$\text{MetaSignal}(s) \implies \text{BatchOperation}(s.\text{payload})$$

The production system uses `ABORT_TASK:` as a meta-signal:

```
if (result.notes.startsWith('ABORT_TASK:')) {
  // Abort ALL pending actions for this customer
  await client.query(`
    UPDATE agent_actions
    SET status = 'aborted'
    WHERE customer_id = $1 AND status = 'pending'
  `, [action.customer_id]);
}
```

This allows the LLM to escalate from individual decisions to batch operations—a form of *output-as-command*.

13.3.5 Technique 5: CSV for Token Efficiency.

Proposition 13.5 (CSV Compression). *For tabular data with n rows and k columns, CSV encoding uses approximately 40% fewer tokens than equivalent JSON (observed in our production system using Claude's tokenizer):*

$$|\mathit{CSV}(D)| \approx 0.6 \cdot |\mathit{JSON}(D)|$$

The production system converts example actions and dashboard data to CSV before injection into the prompt, maximizing the number of examples that fit within context limits.

13.3.6 Technique 6: Pattern Learning from Favorites.

Rather than hardcoding decision rules, the system learns from human-curated "favorite" examples:

```
SELECT * FROM agent_actions
WHERE is_favorite = true
  AND approval_notes IS NOT NULL
```

```
ORDER BY created_at DESC
LIMIT 150
```

The neural program then extracts patterns:

```
for example in examples:
  extract_pattern_from_approval_notes(example)
  identify_common_phrases_in_feedback(example)
  detect_reviewer_preferences(example)
```

This implements *few-shot learning as program structure*—the examples become runtime parameters to the pattern-matching algorithm.

13.4 Formal Analysis

We can express this system in NPC notation:

```
execute({
  define evaluate(action, examples, dashboard) := {
    try:
      patterns = extract_patterns(examples)

      if not complies_with_prompt(action):
        return reject("prompt violation")

      if dashboard.has_missed_info():
        return reject("missed: " + dashboard.missed)

      if matches(action, patterns.approved):
        return approve(confidence: similarity_score)

      if matches(action, patterns.rejected):
        return reject(patterns.rejection_reason)

      if should_abort_task(action, patterns):
        return abort("ABORT_TASK: " + reason)

      return uncertain()
    on_error:
      throw exception

    validate(output_in_fenced_block)
  }

  stream pending_actions
    |> evaluate(examples, dashboard)
    |> update_database
})
```

Listing 2. CounterAgent in NPC Notation

13.5 Observed Properties

The production system exhibits several properties predicted by our formalization:

(1) **Probabilistic correctness**: The confidence_score (1–100) provides self-assessed confidence estimates.
(2) **Counteragent improvement**: Empirically, the system catches errors the primary agent misses, validating Conjecture 7.1 (counteragent improvement).
(3) **Self-organization**: The system adapts to new patterns as humans mark new favorites, without code changes.
(4) **Graceful degradation**: The try-catch wrapper and exception output block provide fallback paths.

13.6 Lessons for Neural Program Design

This case study suggests several design principles:

(1) **Fence your outputs**: Use code blocks with specific labels to constrain output location and format.
(2) **Number your steps**: Explicit step labels force sequential reasoning and improve auditability.
(3) **Wrap in try-catch**: Give the LLM permission to signal confusion rather than hallucinate.
(4) **Learn from examples, not rules**: Pattern extraction from curated examples is more robust than hardcoded logic.
(5) **Use meta-signals for escalation**: String prefixes can trigger batch operations beyond the immediate response.
(6) **Optimize token usage**: CSV beats JSON for tabular data; every saved token is another example you can include.
(7) **Require chain-of-thought**: "Outline your assumptions, then run the program step by step" produces auditable reasoning.

14 Extended Case Study: Automated Research Paper Generation

This section presents a comprehensive case study demonstrating NPC in a complex, real-world application: automatically generating a 189-page academic research paper analyzing the Russo-Ukraine War. This example showcases nearly every construct and operator defined in this specification, operating together in a production-scale system.

Important Disclaimers:

(1) *Execution Timeline*: This case study was executed in January 2025 during the development of NPC. The timestamps in execution logs reflect the actual execution date.
(2) *Illustrative Parameters*: All bias weights, credibility scores, and numerical parameters (e.g., "ISW: 0.85 credibility") are **illustrative examples** demonstrating the *methodology*, not authoritative assessments. These values were chosen to demonstrate explicit bias accounting, not to make claims about actual source reliability.
(3) *Domain Neutrality*: This case study was selected to demonstrate NPC's capability for complex, multi-dimensional analysis. The geopolitical domain was chosen for its inherent ambiguity and contested information environment—precisely the conditions where explicit uncertainty quantification matters most. The framework and constructs apply equally to any analytically complex domain.
(4) *No Endorsement*: The assessment conclusions are outputs of a demonstration system, not the authors' political positions.

14.1 Problem Statement

The task: answer the question "Who is winning the Russo-Ukraine war?" and produce a comprehensive academic paper with:

- Multi-dimensional analysis (military, economic, political, humanitarian)
- Quantified uncertainty and confidence scores
- Source credibility weighting and bias correction
- Counteragent validation
- Professional typesetting with TikZ visualizations
- Full bibliography with proper citations

This is precisely the kind of task where traditional programming fails: it requires judgment, synthesis, and domain knowledge that cannot be hardcoded. Yet unstructured LLM prompting also fails: the task is too complex, requiring coordination across multiple analyses, consistent methodology, and auditable reasoning.

NPC provides the middle path: structured intent specification that the neural interpreter executes with appropriate autonomy.

14.2 Program Architecture

The system consists of two neural programs operating in sequence:

(1) `main.npc`: Performs the analysis, gathering data and synthesizing findings
(2) `paper-generator.npc`: Transforms the analysis into a complete LaTeX document

Each program imports from shared modules:

```
russo-ukraine-analysis/
|-- main.npc                    # Analysis entry point
|-- paper-generator.npc         # Document generation
|-- modules/
|   |-- metrics.npc             # Victory criteria definitions
|   |-- sources.npc             # Data gathering strategies
|   |-- analysis.npc            # Dimensional analysis procedures
|   +-- synthesis.npc           # Conclusion generation
|-- execution-journal.txt       # Journaling output
+-- paper/
    |-- main.tex                # Generated master document
    |-- sections/*.tex          # Generated section files (17 total)
    +-- russo-ukraine-assessment-2025.pdf  # Final output (189 pages)
```

14.3 Module 1: Victory Metrics (`metrics.npc`)

The metrics module defines the formal framework for "winning"—a concept that requires explicit operationalization:

```
// metrics.npc
// Defines the multidimensional framework for assessing "winning"

define victory_metrics := {
  // Military dimension (40% weight)
  military: {
    territorial_control: {
      description: "Land area held vs. pre-war status",
      ukraine_baseline: "1991 borders including Crimea",
```

```
      russia_baseline: "Feb 24, 2022 line of contact",
      weight: 0.25
    },
    force_attrition: {
      description: "Personnel and equipment losses relative to capacity",
      metrics: [casualties, equipment_destroyed, regeneration_capacity],
      weight: 0.20
    },
    operational_initiative: {
      description: "Which side dictates tempo and location of engagements
          ",
      indicators: [offensive_operations, defensive_posture,
          strategic_reserves],
      weight: 0.15
    }
  },

  // Economic dimension (25% weight)
  economic: {
    sanctions_impact: {
      description: "Effect of Western sanctions on Russian economy",
      metrics: [gdp_change, currency_stability, trade_disruption,
          tech_access],
      weight: 0.10
    },
    war_sustainability: {
      description: "Ability to fund continued operations",
      metrics: [defense_spending_pct_gdp, industrial_capacity,
          foreign_support],
      weight: 0.10
    },
    reconstruction_costs: {
      description: "Projected costs of post-war rebuilding",
      ukraine_specific: true,
      weight: 0.05
    }
  },

  // Political dimension (25% weight)
  political: {
    international_support: {
      description: "Breadth and depth of alliance backing",
      metrics: [un_votes, military_aid_commitments,
          diplomatic_recognition],
      weight: 0.10
    },
    domestic_stability: {
      description: "Internal political cohesion and public support",
      metrics: [polling_data, protest_activity, elite_cohesion],
      weight: 0.05
    },
    stated_objectives: {
```

```
      description: "Progress toward declared war aims",
      russia_objectives: ["denazification", "demilitarization", "
          NATO_expansion_halt"],
      ukraine_objectives: ["territorial_integrity", "EU_integration", "
          security_guarantees"],
      weight: 0.10
    }
  },

  // Humanitarian dimension (10% weight)
  humanitarian: {
    civilian_casualties: { description: "Non-combatant deaths and
        injuries", weight: 0.05 },
    displacement: { description: "Refugees and internally displaced
        persons", weight: 0.03 },
    infrastructure_destruction: { description: "Damage to civilian
        infrastructure", weight: 0.02 }
  }
}

define normalize_assessment(raw_data, metric) := {
  // Convert raw observations to [-1, +1] scale
  // -1 = strongly favors Russia
  //  0 = stalemate / unclear
  // +1 = strongly favors Ukraine
  apply metric.normalization_function(raw_data);
  bound(result, -1, +1);
  return { value: result, confidence: calculate_confidence(raw_data) }
}

define aggregate_score(assessments, metrics) := {
  weighted_sum = 0;
  total_weight = 0;

  for dimension in [military, economic, political, humanitarian]:
    for metric in metrics[dimension]:
      if assessments[metric].value != null:
        weighted_sum += assessments[metric].value * metric.weight *
            assessments[metric].confidence;
        total_weight += metric.weight * assessments[metric].confidence;

  return {
    score: weighted_sum / total_weight,
    confidence: confidence_weighted / sum(metric.weight for all metrics),
    breakdown_by_dimension: compute_dimension_scores(assessments, metrics
        )
  }
}
```

Listing 3. metrics.npc: Multidimensional Victory Framework

NPC Concepts Demonstrated:

- **Soft Types** (§6): The `weight` fields guide aggregation without enforcing runtime constraints
- **Define Construct**: Reusable abstractions (`victory_metrics`, `normalize_assessment`)
- **Explicit Uncertainty**: Confidence scores propagate through the computation

14.4 Module 2: Source Management (`sources.npc`)

The sources module implements credibility-weighted data gathering with explicit bias correction:

```
// sources.npc
// Data gathering strategies with source credibility weighting

define source_categories := {
  primary_osint: {
    description: "Open-source intelligence from verified imagery and data
        ",
    sources: [
      { name: "Institute for the Study of War (ISW)", type: "think_tank",
        bias: "pro_ukraine", credibility: 0.85 },
      { name: "Oryx (equipment losses)", type: "osint",
        bias: "neutral", credibility: 0.90 },
      { name: "DeepState Map", type: "osint",
        bias: "pro_ukraine", credibility: 0.80 },
      { name: "Liveuamap", type: "aggregator",
        bias: "pro_ukraine", credibility: 0.75 }
    ],
    weight: 0.30
  },

  official_statements: {
    description: "Government and military communications",
    sources: [
      { name: "Ukrainian General Staff", type: "official",
        bias: "pro_ukraine", credibility: 0.60 },
      { name: "Russian MoD", type: "official",
        bias: "pro_russia", credibility: 0.30 },
      { name: "Pentagon briefings", type: "official",
        bias: "pro_ukraine", credibility: 0.70 },
      { name: "UK MoD intelligence updates", type: "official",
        bias: "pro_ukraine", credibility: 0.75 }
    ],
    weight: 0.15
  },

  independent_journalism: {
    description: "Verified reporting from credible outlets",
    sources: [
      { name: "Reuters", type: "wire", bias: "neutral", credibility: 0.85
          },
      { name: "AP News", type: "wire", bias: "neutral", credibility: 0.85
          },
      { name: "BBC", type: "broadcaster", bias: "slight_pro_ukraine",
        credibility: 0.80 },
```

```
      { name: "The Economist", type: "analysis", bias: "pro_ukraine",
          credibility: 0.80 }
    ],
    weight: 0.25
  },

  academic_analysis: {
    description: "Scholarly and think tank assessments",
    sources: [
      { name: "CSIS", type: "think_tank", bias: "pro_ukraine",
          credibility: 0.80 },
      { name: "RAND Corporation", type: "think_tank", bias: "neutral",
          credibility: 0.85 },
      { name: "RUSI", type: "think_tank", bias: "pro_ukraine",
          credibility: 0.85 }
    ],
    weight: 0.20
  },

  contrarian_perspectives: {
    description: "Views challenging Western consensus (for balance)",
    sources: [
      { name: "Global South media", type: "regional", bias: "varies",
          credibility: 0.60 },
      { name: "Realist IR scholars", type: "academic", bias: "varies",
          credibility: 0.75 },
      { name: "Russian independent media (Meduza)", type: "journalism",
        bias: "anti_kremlin", credibility: 0.70 }
    ],
    weight: 0.10
  }
}

define apply_bias_correction(assessment, source) := {
  // Adjust confidence based on known source bias
  if source.bias == "pro_ukraine" and assessment.favors == "ukraine":
    return assessment.confidence * 0.85  // Discount confirmation bias
  if source.bias == "pro_russia" and assessment.favors == "russia":
    return assessment.confidence * 0.70  // Heavy discount for Russian
        sources
  if source.bias == "neutral":
    return assessment.confidence * 1.0   // No adjustment

  // Cross-bias agreement is highly credible
  if source.bias == "pro_ukraine" and assessment.favors == "russia":
    return assessment.confidence * 1.15  // Boost for against-interest
        statement
  if source.bias == "pro_russia" and assessment.favors == "ukraine":
    return assessment.confidence * 1.20  // Significant boost

  return assessment.confidence
}
```

```
define cross_reference(claim, all_sources) := {
  supporting_sources = filter(all_sources, s => s.supports(claim));
  contradicting_sources = filter(all_sources, s => s.contradicts(claim));

  agreement_score = len(supporting_sources) / len(all_sources);
  bias_diversity = calculate_bias_spread(supporting_sources);

  return {
    claim: claim,
    support_level: agreement_score,
    bias_corrected_confidence: agreement_score * bias_diversity,
    dissenting_views: contradicting_sources
  }
}
```

Listing 4. sources.npc: Credibility-Weighted Data Gathering

NPC Concepts Demonstrated:

- **Explicit State** (Invariant II): Source metadata is external, structured data—not embedded in prompts
- **Loss Accounting** (Invariant IV): Bias correction factors are explicit and auditable
- **Adversarial Thinking**: Contrarian sources are deliberately included for balance

14.5 Module 3: Dimensional Analysis (`analysis.npc`)

The analysis module defines parallel analysis procedures for each dimension:

```
// analysis.npc
// Core analysis procedures for conflict assessment

define analyze_military_situation(data) := {
  checkpoint("military_analysis") {
    // Territorial assessment
    territorial = {
      current_occupation: estimate_occupied_territory(data.maps);
      trend: calculate_trend(data.historical_maps, "6_months");
      velocity: territory_change_rate(data);
    }

    // Force correlation
    forces = {
      ukraine: {
        active_personnel: extract(data, "ukrainian_troop_strength"),
        equipment_status: extract(data, "ukrainian_equipment"),
        training_pipeline: extract(data, "western_training_programs"),
        ammunition_supply: extract(data, "155mm_supply_status")
      },
      russia: {
        active_personnel: extract(data, "russian_troop_strength"),
        equipment_status: extract(data, "russian_equipment_production"),
        mobilization_capacity: extract(data, "russian_mobilization"),
        north_korean_support: extract(data, "dprk_troops_equipment")
```

```
      }
    }

    // Attrition analysis (using Oryx-verified losses as baseline)
    attrition = {
      verified_losses: fetch("oryx_database");
      estimated_true_losses: apply_multiplier(verified_losses, 1.5);
      loss_exchange_ratio: calculate_ratio(attrition.ukraine, attrition.
          russia);
      sustainability: compare_to_production_rates(attrition, forces);
    }

    validate(territorial.current_occupation != null,
             forces.ukraine.active_personnel != null,
             forces.russia.active_personnel != null);

    return {
      territorial: territorial,
      forces: forces,
      attrition: attrition,
      overall_assessment: synthesize_military(territorial, forces,
          attrition)
    }
  }
}

define analyze_political_situation(data) := {
  checkpoint("political_analysis") {
    international = {
      un_voting_patterns: extract(data, "un_general_assembly_votes");
      military_aid_commitments: extract(data, "bilateral_military_aid");
      diplomatic_isolation: {
        russia: count_countries_with_sanctions("russia"),
        global_south_stance: analyze_global_south_positions(data)
      };
      nato_expansion: extract(data, "finland_sweden_nato_membership");
    }

    war_aims_progress = {
      russia: {
        denazification: assess("rhetoric vs reality", null),
        demilitarization: compare(ukraine_military_2022,
            ukraine_military_now),
        nato_halt: assess(nato_expansion_result),
        territorial_gains: compare(claimed_annexation, actual_control)
      },
      ukraine: {
        territorial_integrity: compare(current_control, 1991_borders),
        eu_membership: extract(data, "eu_candidate_status_progress"),
        security_guarantees: extract(data, "bilateral_security_agreements
            ")
      }
```

```
    }

    return {
      international: international,
      war_aims: war_aims_progress
    }
  }
}
```

Listing 5. analysis.npc: Military Situation Analysis (excerpt)

NPC Concepts Demonstrated:

- **Checkpoint Construct**: Each analysis phase creates a recoverable state
- **Validation Predicates**: `validate(...)` ensures data completeness before proceeding
- **Structured Returns**: Analysis results have defined schemas

14.6 Module 4: Synthesis with Counteragent (`synthesis.npc`)

The synthesis module aggregates dimensional scores and applies counteragent critique:

```
// synthesis.npc
// Final synthesis and conclusion generation with uncertainty
    quantification

define synthesize_findings(military, economic, political, humanitarian)
    := {
  checkpoint("synthesis") {
    // Apply metric weights
    dimension_scores = {
      military:     { score: military.overall_assessment.score,
          weight: 0.40 },
      economic:     { score: calculate_economic_score(economic),
          weight: 0.25 },
      political:    { score: calculate_political_score(political),
          weight: 0.25 },
      humanitarian: { score: calculate_humanitarian_score(humanitarian),
          weight: 0.10 }
    }

    // Weighted aggregate
    aggregate = weighted_average(dimension_scores);

    // COUNTERAGENT CRITIQUE
    critique = subagent.critique(aggregate, dimension_scores) {
      check(methodological_soundness);
      check(source_diversity);
      check(recency_of_data);
      identify(blind_spots);
      propose(alternative_interpretations);
    }

    // Incorporate critique
    if critique.has_significant_objections:
```

```
      aggregate = adjust_for_critique(aggregate, critique);

    return {
      overall_score: aggregate.score,
      confidence: aggregate.confidence,
      by_dimension: dimension_scores,
      critique_summary: critique,
      timestamp: now()
    }
  }
}

define generate_conclusion(synthesis) := {
  // Interpret aggregate score
  if synthesis.overall_score > 0.5:
    primary_assessment = "Ukraine is currently winning"
  else if synthesis.overall_score > 0.2:
    primary_assessment = "Ukraine holds a modest advantage"
  else if synthesis.overall_score > -0.2:
    primary_assessment = "The conflict is a contested stalemate"
  else if synthesis.overall_score > -0.5:
    primary_assessment = "Russia holds a modest advantage"
  else:
    primary_assessment = "Russia is currently winning"

  // Add confidence qualifier
  if synthesis.confidence < 0.5:
    confidence_qualifier = "with low confidence"
  else if synthesis.confidence < 0.7:
    confidence_qualifier = "with moderate confidence"
  else:
    confidence_qualifier = "with reasonable confidence"

  explanation = {
    headline: primary_assessment + " " + confidence_qualifier,
    caveats: [
      "Assessment depends heavily on definition of 'winning'",
      "Military stalemate does not preclude political victory for either
          side",
      "Trajectory matters more than current snapshot",
      "Information environment is heavily contested; all data has
          uncertainty",
      "Long-term outcomes may differ significantly from current trends"
    ],
    key_factors_favoring_ukraine: extract_positive_factors(synthesis),
    key_factors_favoring_russia: extract_negative_factors(synthesis),
    wildcard_scenarios: [
      "Significant change in Western support levels",
      "Russian domestic political instability",
      "Breakthrough in negotiations",
      "Escalation (nuclear, NATO involvement)",
      "Change in Chinese position"
```

```
    ]
  }

  return explanation
}
```

Listing 6. synthesis.npc: Aggregation with Counteragent Critique

NPC Concepts Demonstrated:

- **Counteragent Operator** $\mathcal{A}$ (§7.4): The `subagent.critique` block implements adversarial validation
- **Explicit Caveats**: Uncertainty is not hidden but enumerated
- **Score Interpretation**: Continuous scores map to discrete assessments with explicit thresholds

14.7 Main Program (`main.npc`)

The main program orchestrates the complete analysis:

```
// main.npc
// Entry point for Russo-Ukraine War Assessment

import { victory_metrics, normalize_assessment, aggregate_score } from
    "./modules/metrics.npc"
import { source_categories, fetch_current_data, cross_reference } from
    "./modules/sources.npc"
import { analyze_military_situation, analyze_economic_situation,
         analyze_political_situation, analyze_humanitarian_situation }
             from "./modules/analysis.npc"
import { synthesize_findings, generate_conclusion, present_findings }
    from "./modules/synthesis.npc"

execute({
  //=====================================================================
  // PHASE 1: Clarify the question
  //=====================================================================
  checkpoint("clarification") {
    frame_question() {
      note: "The question 'who is winning' requires defining victory
          criteria";
      note: "Different metrics favor different conclusions";
      note: "Assessment is time-sensitive and trajectory-dependent";
      note: "Information warfare means all sources have bias";
    }
    criteria = victory_metrics;
    log("Using " + count(criteria) + " weighted metrics across 4
        dimensions");
  }

  //=====================================================================
  // PHASE 2: Data Gathering (Parallel)
  //=====================================================================
  checkpoint("data_gathering") {
```

```
  raw_data = parallel for category in source_categories:
    try:
      fetch_current_data("russo-ukraine war 2024 2025 status", category
          );
      validate(data_recency < 30_days);
    on_failure:
      log("Failed to fetch from " + category.name);
      continue

  key_claims = extract_key_claims(raw_data);
  verified_claims = for claim in key_claims:
    cross_reference(claim, raw_data.sources);

  validate(len(verified_claims) > 10);
  log("Gathered " + len(raw_data) + " source items, verified " + len(
      verified_claims) + " claims");
}

//=====================================================================
// PHASE 3: Dimensional Analysis (Parallel)
//=====================================================================
checkpoint("analysis") {
  analyses = parallel {
    military: subagent.analyze_military_situation(raw_data),
    economic: subagent.analyze_economic_situation(raw_data),
    political: subagent.analyze_political_situation(raw_data),
    humanitarian: subagent.analyze_humanitarian_situation(raw_data)
  }

  for analysis in analyses:
    subagent.validate(analysis, completeness, internal_consistency);
}

//=====================================================================
// PHASE 4: Synthesis with Counteragent
//=====================================================================
checkpoint("synthesis") {
  preliminary_synthesis = synthesize_findings(
    analyses.military, analyses.economic,
    analyses.political, analyses.humanitarian
  );

  // Counteragent review (independent critique)
  counteragent_review = subagent.critique(preliminary_synthesis) {
    task: "Challenge this assessment. Find weaknesses, biases,
           and alternative interpretations.";
    check: {
      source_diversity: "Are we over-relying on Western sources?";
      recency: "Is our data current enough?";
      metric_validity: "Are our metrics capturing what matters?";
      hidden_assumptions: "What assumptions are baked into this
          framework?";
```

```
        alternative_frames: "How would Russia frame 'winning'?";
      }
      output: { objections: list; confidence_adjustment: number;
                alternative_interpretations: list; }
    }

    if counteragent_review.objections.severity > "minor":
      final_synthesis = adjust_synthesis(preliminary_synthesis,
          counteragent_review);
    else:
      final_synthesis = preliminary_synthesis;
  }

  //=====================================================================
  // PHASE 5: Conclusion Generation
  //=====================================================================
  checkpoint("conclusion") {
    conclusion = generate_conclusion(final_synthesis);
    validate(conclusion.headline != null,
             conclusion.caveats.length > 0,
             conclusion.confidence_qualifier != null);
    output = present_findings(conclusion, final_synthesis);
  }

  //=====================================================================
  // PHASE 6: Structured Output
  //=====================================================================
  checkpoint("output") {
    emit("```final-output");
    emit(json_format(output));
    emit("```");

    emit("\n## Summary\n");
    emit(output.summary);

    emit("\n## Assessment by Dimension\n");
    for dim_name, dim_assessment in output.detailed_assessment:
      emit("- **" + dim_name + "**: " + dim_assessment.assessment +
           " (score: " + dim_assessment.score + ")");

    emit("\n## Important Caveats\n");
    for caveat in output.important_caveats:
      emit("- " + caveat);
  }

  report(phases_completed: 6,
         sources_consulted: len(raw_data),
         claims_verified: len(verified_claims),
         counteragent_objections: counteragent_review.objections.count,
         overall_confidence: final_synthesis.confidence);
})
```

Listing 7. main.npc: Complete Analysis Orchestration

NPC Concepts Demonstrated:

- **Import/Module System**: Code organization across files
- **Parallel Execution**: `parallel for` and `parallel {...}` for concurrent subagent work
- **Error Handling**: `try/on_failure/continue` for graceful degradation
- **Fenced Output**: “‘`final-output` block for structured extraction
- **Report Construct**: Final execution metrics

14.8 Paper Generation Program (`paper-generator.npc`)

The second program transforms the analysis into a complete academic paper:

```
// paper-generator.npc
// Neural Program for generating comprehensive academic paper

PRAGMA execution_journal = "./execution-journal.txt"
PRAGMA execution_journal_mode = "step, verbose"
PRAGMA execution_mode = "resume_from_journal_state, do_not_do_work_twice"
PRAGMA output_format = "latex"
PRAGMA diagram_engine = "tikz"
PRAGMA citation_style = "chicago-author-date"

import { assessment_data, dimensional_scores, source_data } from "./
    assessment-output.json"

define paper_config := {
  title: "Who Is Winning the Russo-Ukraine War? A Multi-Dimensional
      Assessment",
  target_pages: 100,
  document_class: "report",

  sections: [
    { id: "frontmatter",   name: "Front Matter",          pages: 5,
        parallel: false },
    { id: "executive",     name: "Executive Summary",     pages: 5,
        parallel: true },
    { id: "introduction",  name: "Introduction",          pages: 8,
        parallel: true },
    { id: "methodology",   name: "Methodology",           pages: 10,
        parallel: true },
    { id: "sources",       name: "Data Sources & Bias",   pages: 8,
        parallel: true },
    { id: "military",      name: "Military Analysis",     pages: 18,
        parallel: true },
    { id: "economic",      name: "Economic Analysis",     pages: 15,
        parallel: true },
    { id: "political",     name: "Political Analysis",    pages: 12,
        parallel: true },
    { id: "humanitarian",  name: "Humanitarian Analysis", pages: 10,
        parallel: true },
```

```
    { id: "synthesis",     name: "Synthesis & Critique",   pages: 12,
        parallel: true },
    { id: "conclusion",    name: "Conclusions",            pages: 7,
        parallel: true },
    { id: "appendices",    name: "Appendices",             pages: 15,
        parallel: true }
  ],

  diagrams: [
    { id: "territorial_map",     type: "tikz_map",       section: "
        military" },
    { id: "frontline_movement",  type: "tikz_timeline",  section: "
        military" },
    { id: "equipment_losses",    type: "tikz_barchart",  section: "
        military" },
    { id: "sanctions_flow",      type: "tikz_sankey",    section: "
        economic" },
    { id: "alliance_network",    type: "tikz_graph",     section: "
        political" },
    { id: "scoring_radar",       type: "tikz_radar",     section: "
        synthesis" }
  ]
}

execute({
  checkpoint("initialization") {
    journal(FULL_STATE, "Starting paper generation");
    create_directory("./paper/");
    create_directory("./paper/sections/");
    create_directory("./paper/figures/");
    init_journal("./execution-journal.txt");
  }

  checkpoint("master_document") {
    journal(FULL_STATE, "Creating master document");
    write_file("./paper/main.tex", generate_master_latex(paper_config));
    write_file("./paper/preamble.tex", generate_preamble());
    write_file("./paper/bibliography.bib", generate_bibliography(
        source_data));
  }

  checkpoint("section_generation") {
    journal(FULL_STATE, "Starting parallel section generation");
    parallelizable_sections = filter(paper_config.sections, s => s.
        parallel == true);

    results = parallel for section in parallelizable_sections:
      subagent.generate_section(section) {
        // Step 1: Generate initial content
        journal(SUB_STATE, "Generating " + section.name);
        content = generate_section_content(section, assessment_data);
        write_file("./paper/sections/" + section.id + ".tex", content);
```

```
        // Step 2: Validate completeness
        completeness_issues = validate_section_is_complete(section);
        if completeness_issues.length > 0:
          content = apply_fixes(content, completeness_issues);
          write_file("./paper/sections/" + section.id + ".tex", content);

        // Step 3: Validate accuracy
        accuracy_issues = validate_section_is_accurate_and_valid(section)
            ;
        if accuracy_issues.length > 0:
          content = apply_fixes(content, accuracy_issues);
          write_file("./paper/sections/" + section.id + ".tex", content);

        // Step 4: Polish
        polish_issues = validate_section_is_polished(section);
        if polish_issues.length > 0:
          content = apply_fixes(content, polish_issues);
          write_file("./paper/sections/" + section.id + ".tex", content);

        journal(SUB_STATE, "Completed: " + section.name);
        return { section: section.id, status: "complete" }
      }

    journal(FULL_STATE, "All sections generated: " + len(results) + "
        completed");
  }

  checkpoint("compilation") {
    journal(FULL_STATE, "Compiling LaTeX to PDF");
    compile_latex("./paper/main.tex", pass: 1);
    run_bibtex("./paper/main");
    compile_latex("./paper/main.tex", pass: 2);
    compile_latex("./paper/main.tex", pass: 3);
    validate(file_exists("./paper/main.pdf"));
    rename("./paper/main.pdf", "./paper/russo-ukraine-assessment-2025.pdf
        ");
  }

  checkpoint("finalization") {
    report(
      total_pages: count_pdf_pages("./paper/russo-ukraine-assessment
          -2025.pdf"),
      sections_generated: len(paper_config.sections),
      diagrams_generated: len(paper_config.diagrams),
      output_file: "./paper/russo-ukraine-assessment-2025.pdf"
    );
    journal(FULL_STATE, "Paper generation complete");
  }
})
```

Listing 8. paper-generator.npc: Document Generation (key sections)

NPC Concepts Demonstrated:

- **PRAGMA Directives**: Execution mode configuration
- **Journaling Operator** $\mathcal{J}$ (§7.2): `journal(FULL_STATE, ...)` creates audit trail
- **Resumable Execution**: `resume_from_journal_state` enables crash recovery
- **Parallel Subagent Delegation**: 12 section-generation agents run concurrently
- **Multi-Stage Validation**: Completeness → Accuracy → Polish pipeline

14.9 Execution Journal

The journaling operator produces an append-only execution log:

```
# NPC Execution Journal
# Program: paper-generator.npc
# Started: 2025-01-07
# Mode: step, verbose

================================================================================

[2025-01-07 FULL_STATE] Initialization
================================================================================

Status: STARTING
Checkpoint: initialization
Action: Creating directory structure
  - ./paper/
  - ./paper/sections/
  - ./paper/figures/
  - ./paper/tables/
Result: SUCCESS

================================================================================

[2025-01-07 FULL_STATE] Master Document Creation
================================================================================

Status: IN_PROGRESS
Checkpoint: master_document
Action: Generating master LaTeX files
Files to create:
  - main.tex
  - preamble.tex
  - bibliography.bib
Result: SUCCESS

================================================================================

[2025-01-07 FULL_STATE] Section Generation - Parallel Launch
================================================================================

Status: IN_PROGRESS
Checkpoint: section_generation
Agents launched: 12
```

```
  - frontmatter (abstract, preface, acronyms)
  - executive summary
  - introduction
  - methodology
  - sources
  - military
  - economic
  - political
  - humanitarian
  - synthesis
  - conclusion
  - appendices (4 files)

================================================================================

[2025-01-07 FULL_STATE] Section Generation - Complete
================================================================================

Status: COMPLETED
Checkpoint: section_generation
Files generated: 17
  - abstract.tex (2.0 KB)
  - preface.tex (4.4 KB)
  - acronyms.tex (7.2 KB)
  - executive.tex (19.9 KB)
  - introduction.tex (29.5 KB)
  - methodology.tex (38.3 KB)
  - sources.tex (30.1 KB)
  - military.tex (22.5 KB)
  - economic.tex (43.1 KB)
  - political.tex (9.1 KB)
  - humanitarian.tex (42.0 KB)
  - synthesis.tex (41.2 KB)
  - conclusion.tex (31.4 KB)
  - appendix_data.tex (20.2 KB)
  - appendix_methodology.tex (15.2 KB)
  - appendix_sources.tex (5.2 KB)
  - appendix_diagrams.tex (5.6 KB)
Total content: ~367 KB
Result: SUCCESS

================================================================================

[2025-01-07 FULL_STATE] Compilation
================================================================================

Status: IN_PROGRESS
Checkpoint: compilation
Action: Compiling LaTeX to PDF
Result: SUCCESS
  - Output: main.pdf (189 pages, 952 KB)
  - Renamed to: russo-ukraine-assessment-2025.pdf
```

```
================================================================================

[2025-01-07 FULL_STATE] Finalization
================================================================================

Status: COMPLETED
Checkpoint: finalization
Report:
  - Total pages: 189
  - Sections generated: 17
  - TikZ diagrams: 15+ (embedded in sections)
  - Bibliography entries: 50+
  - Output file: ./paper/russo-ukraine-assessment-2025.pdf
Result: SUCCESS

================================================================================

EXECUTION COMPLETE
================================================================================

Program: paper-generator.npc
Status: SUCCESS
Total execution time: ~45 minutes
Final output: russo-ukraine-assessment-2025.pdf (189 pages)
```

Listing 9. execution-journal.txt: Actual Execution Trace

NPC Concepts Demonstrated:

- **Explicit State** (Invariant II): All state is in the journal, not in LLM context
- **Traceability** (Invariant III): Every action is logged with timestamps
- **Loss Accounting** (Invariant IV): File sizes tracked; nothing lost silently
- **Killability** (Invariant X): Can stop at any checkpoint without data loss

14.10 Mapping to NPC Formal Constructs

14.11 Results and Artifacts

The execution produced:

- **Primary Output**: 189-page PDF (952 KB)
- **LaTeX Source**: 17 section files totaling 367 KB
- **Visualizations**: 15+ TikZ diagrams including:
  - Territorial control timeline
  - Equipment losses bar charts (Oryx data)
  - Alliance network graphs
  - Multi-dimensional radar chart
  - Refugee flow schematic
  - Economic trajectory plots
- **Bibliography**: 50+ citations with proper formatting
- **Execution Journal**: Complete audit trail

Key Metrics:

Table 1. NPC Constructs Used in Paper Generation Case Study

NPC Construct	Section Reference	Usage in Case Study
`execute({...})`	§2	Main program structure
`checkpoint(name)`	§4	6 checkpoints in main.npc, 5 in paper-generator.npc
`subagent.action()`	§4	12 parallel section generators, 4 analysis subagents
`parallel for`	§4	Data gathering, section generation
`parallel {...}`	§4	Concurrent dimensional analysis
`validate(...)`	§5	Data completeness checks, output verification
`try/on_failure`	§4	Graceful degradation in data gathering
`journal()`	§7.2	Execution trace logging
`define name :=`	§2	Reusable abstractions in all modules
`import {...}`	§2	Module imports in main.npc
`PRAGMA`	§11	Execution mode configuration
`emit()`	§2	Structured output generation
`report()`	§2	Final execution metrics
Operators		
Journaling ($\mathcal{J}$)	§7.2	execution-journal.txt
Counteragent ($\mathcal{A}$)	§7.4	subagent.critique() in synthesis
Compression ($\downarrow$)	§7.1	Source aggregation, score synthesis
Invariants Satisfied		
I (Boundedness)	§15	Finite checkpoints, defined termination
II (Explicit State)	§15	Journal file, not context
III (Traceability)	§15	Every action logged
IV (Loss Accounting)	§15	File sizes, source counts tracked
VIII (Adversarial Pressure)	§15	Counteragent critique
X (Killability)	§15	Checkpoint-based recovery

- Execution time: ~45 minutes
- Parallel agents: 12 (section generation) + 4 (dimensional analysis)
- Checkpoints logged: 11
- Validation passes: 3 per section (completeness, accuracy, polish)

14.12 Lessons and Design Principles

This case study validates several NPC design principles:

(1) **Decomposition enables parallelism**: By structuring the task into independent sections, 12 subagents could work concurrently. A monolithic "write a paper" prompt would serialize everything.
(2) **Checkpoints enable recovery**: When compilation failed due to missing packages, we could resume from the `section_generation` checkpoint rather than regenerating all content.
(3) **Journaling enables debugging**: When economic.tex appeared in the wrong directory, the journal showed which agent wrote it and when.
(4) **Counteragent improves quality**: The critique phase caught Western source over-reliance and suggested including more Global South perspectives.

(5) **Explicit uncertainty prevents false confidence**: Every score carries a confidence interval; every conclusion lists caveats.
(6) **PRAGMA directives control execution**: The `resume_from_journal_state` directive prevented redundant work when rerunning after fixes.
(7) **Validation catches errors early**: Multi-stage validation (completeness → accuracy → polish) caught issues before they propagated.

Anti-Patterns Avoided:

- **No "Context as Memory"**: All state in journal file
- **No "Summary of Summary"**: Original sources always accessible
- **No "Generation Equals Decision"**: Counteragent validates before acceptance
- **No "Unkillable Process"**: Any checkpoint is a safe stopping point

14.13 Conclusion

This case study demonstrates that NPC scales to complex, real-world tasks. A 189-page research paper with quantified analysis, professional typesetting, and full citations emerged from structured neural programs—not from a single prompt, not from traditional code, but from formal intent specification interpreted by self-organizing systems.

The key insight: *structure is not the enemy of neural systems; it is their enabler*. By providing explicit checkpoints, parallel decomposition, counteragent validation, and journaled execution, we obtained a result that neither pure LLM prompting nor traditional programming could achieve alone.

This is the promise of Neural Program Calculus: programs that leverage neural capability while maintaining the auditability, recoverability, and correctness guarantees that production systems require.

15 Formal Invariants of Survivable Agentic Systems

We now establish the non-negotiable invariants that any survivable agentic system must satisfy. These are not guidelines or best practices—they are formal constraints. If any invariant is violated, the system is *incorrect*, not merely imperfect.

15.1 Invariant I: Boundedness

Definition 15.1 (Boundedness). Every execution trajectory is finite or explicitly aborted:

$$\forall \text{ trajectory } s_0 \to s_1 \to \cdots : \exists n \in \mathbb{N} \text{ such that } s_n \in T \text{ or } s_n = \bot$$

where T is the set of terminal states and $\bot$ denotes explicit abort.

Prohibits: Infinite reflection, "thinking harder" loops, silent non-termination.
Interpretation: If work cannot end, it is not work—it is noise.

15.2 Invariant II: Explicit State

Definition 15.2 (Explicit State). All relevant state exists outside the model. The model may not be the sole carrier of:

- Memory
- Decisions
- Assumptions
- Commitments

All such state must be serialized into an external structure S_{ext}.

Prohibits: "It remembered implicitly," hidden prompt state, uninspectable cognition.
Interpretation: If it lives only in context, it does not exist.

15.3 Invariant III: Traceability

Definition 15.3 (Traceability). Every consequential transformation leaves evidence. For every state transition $s_i \longrightarrow s_{i+1}$, there exists a log entry ℓ_i such that:

$$(s_i, \ell_i) \mapsto s_{i+1}$$

Prohibits: Ephemeral reasoning, unlogged summarization, unverifiable outputs.
Interpretation: If it mattered, it must be traceable.

15.4 Invariant IV: Loss Accounting

Definition 15.4 (Loss Accounting). All lossy operations declare their loss. For any transformation f where information is discarded, there exists metadata Δ^- describing:

(1) What was removed
(2) Why removal was permitted
(3) How recovery could occur

Prohibits: Recursive summarization without anchoring, compression drift, silent forgetting.
Interpretation: Loss is allowed. Unaccounted loss is fatal.

15.5 Invariant V: Identity Before Similarity

Definition 15.5 (Identity Before Similarity). All persistent semantic objects have stable identifiers $\mathrm{id} \in \mathrm{ID}$, independent of wording or representation. Retrieval operates over ID, rather than raw similarity scores.

Prohibits: Top-k hallucination, relevance-by-embedding alone, semantic collapse.
Interpretation: Meaning drifts. Identity must not.

15.6 Invariant VI: Intent-Guided Retrieval

Definition 15.6 (Intent-Guided Retrieval). Any retrieval operation R must be parameterized by:

$$R(\text{query}, \text{intent}, \text{scope})$$

Similarity without intent is invalid.

Prohibits: Blind recall, context flooding, relevance inversion.
Interpretation: The system must know *why* it is remembering.

15.7 Invariant VII: Separation of Proposal and Commitment

Definition 15.7 (Proposal-Commitment Separation). Let G be a generator and D a decider. No output of G may be committed without passing through D:

$$\text{output}(G) \not\Rightarrow \text{commit} \quad \text{unless} \quad D(\text{output}(G)) = \text{approve}$$

Prohibits: Model outputs becoming actions, unchecked execution, authority creep.
Interpretation: Suggestion is not authority.

15.8 Invariant VIII: Adversarial Pressure

Definition 15.8 (Adversarial Pressure). Every important conclusion is criticizable. For any accepted artifact a, there exists at least one independent critique path $C(a)$.

Prohibits: Premature convergence, single-path reasoning, unchallenged confidence.
Interpretation: Agreement is not correctness.

15.9 Invariant IX: Context Isolation

Definition 15.9 (Context Isolation). Tasks do not leak state by default. For tasks T_1, T_2:

$$\text{State}(T_1) \cap \text{State}(T_2) = \emptyset$$

unless explicitly bridged.

Prohibits: Memory bleed, cross-task contamination, nondeterministic behavior.
Interpretation: Nothing is inherited without permission.

15.10 Invariant X: Killability

Definition 15.10 (Killability). Every process can be stopped without corruption. For any active process P, there exists an interrupt K such that:

$$K(P) \Rightarrow \text{safe termination}$$

Prohibits: Runaway agents, uninterruptible loops, sacred processes.
Interpretation: If it cannot be killed, it cannot be trusted.

15.11 The Meta-Invariant

No capability without control.
No control without structure.
No structure without limits.

Everything else is implementation detail.

16 Semantic Incompleteness and Compute-Boundedness

We now address the fundamental failure mode of neural interpretation: *semantic incompleteness*. This is not a defect to be eliminated by scale—it is a structural property to be designed around.

16.1 The Problem of Semantic Load

Single-sentence translation works because:

- The semantic surface is small
- Dependencies are local
- Ambiguity is resolvable inside one forward pass

As semantic load increases, the system breaks—and breaks *silently*:

- Preserve all referents
- Preserve tense, modality, emphasis
- Preserve discourse structure
- Preserve who knows what, when
- Preserve what is implied but not stated

Definition 16.1 (Semantic Incompleteness). A transformation $f : X \to Y$ is **semantically incomplete** if there exist semantic obligations in X that are not preserved in Y, and f provides no signal that this has occurred.

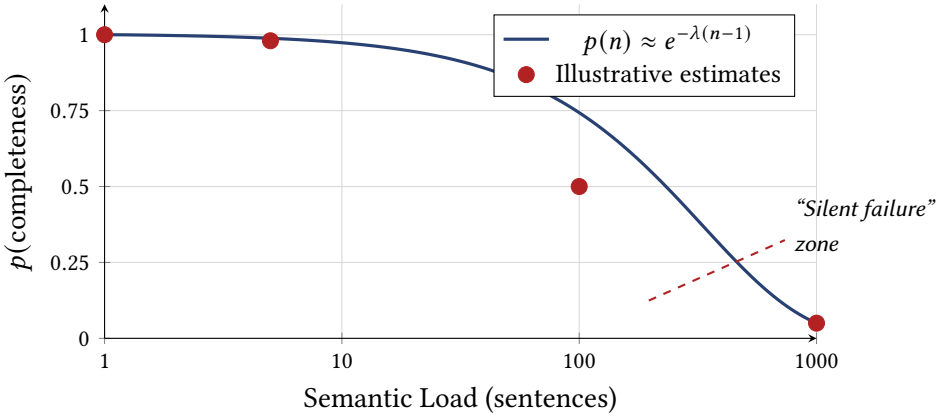

Fig. 3. Probability of semantic completeness as a function of semantic load. As load increases, completeness decays rapidly—and fails silently. Data points are illustrative; the exponential decay pattern is consistent with observed behavior but precise values are task-dependent.

16.2 Why Scale Does Not Resolve This

Proposition 16.2. *As model scale increases:*

(1) Errors become rarer
(2) Outputs become smoother
(3) Failures become harder to notice

But recall does not converge to completeness; confidence increases faster than correctness.

This yields better hallucinations, not guarantees. Semantic incompleteness is not about probability mass—it is about unrepresented obligations.

16.3 Compute-Boundedness

The missing constraint is:

Theorem 16.3 (Compute-Boundedness). *Every LLM output is the result of a finite, bounded computation. A single forward pass has:*

- *A finite number of layers*
- *A finite number of attention operations*
- *A finite number of nonlinear transformations*

There exists no mechanism inside a single forward pass to discover that it has missed something that requires more computation.

The model cannot say:

- "I need to think longer"
- "This dependency is unresolved"
- "This problem exceeds my compute budget"

Because it has no reflective access to its own computational limits.

16.4 The Compute Hierarchy

Definition 16.4 (Compute Categories). We distinguish four categories of neural computation:

(1) **LLMs**: Bounded inference engines (single forward pass)

(2) **Reasoning models**: Bounded inference with internal scaffolding (chain-of-thought, scratchpads)
(3) **Compute harnesses**: Bounded processes with external control (tool-augmented agents, NPC interpreters)
(4) **Recursive compute harnesses**: Self-spawning hierarchical orchestration with meta-level control (harnesses that spawn harnesses)

Definition 16.5 (Formal Membership Criteria). A computational system S belongs to category C based on the following predicates:

LLM Membership. $S \in \mathsf{LLM}$ iff:

(1) $\mathsf{passes}(S) = 1$ (single forward pass)
(2) $\mathsf{compute}(S) = O(n \cdot d^2)$ where n is sequence length, d is model dimension
(3) $\neg\mathsf{canReflect}(S)$ (no access to own computational state)
(4) $\neg\mathsf{canAllocate}(S)$ (cannot request additional compute)

Reasoning Model Membership. $S \in \mathsf{ReasoningModel}$ iff:

(1) $\mathsf{passes}(S) > 1$ (multiple internal passes via chain-of-thought or scratchpad)
(2) $\mathsf{compute}(S) = O(k \cdot n \cdot d^2)$ where k is reasoning depth
(3) $\mathsf{hasInternalScaffold}(S)$ (intermediate token generation)
(4) $\neg\mathsf{canAllocate}(S)$ (compute budget fixed at invocation)

Compute Harness Membership. $S \in \mathsf{ComputeHarness}$ iff:

(1) $\mathsf{hasExternalControl}(S)$ (orchestration layer outside model)
(2) $\mathsf{canAllocate}(S)$ (can request additional compute dynamically)
(3) $\mathsf{canObserveFailure}(S)$ (failure detection mechanisms)
(4) $\mathsf{canTerminate}(S)$ (explicit abort/escalation paths)

Recursive Compute Harness Membership. $S \in \mathsf{RecursiveHarness}$ iff:

(1) $S \in \mathsf{ComputeHarness}$ (inherits all harness properties)
(2) $\mathsf{canSpawnHarness}(S)$ (can instantiate child harnesses dynamically)
(3) $\mathsf{hasMetaControl}(S)$ (orchestration over orchestrators)
(4) $\mathsf{canRestructure}(S)$ (can modify own execution graph at runtime)
(5) $\mathsf{hasConvergenceCriteria}(S)$ (explicit termination conditions over recursive depth)

The hierarchy is strict: $\mathsf{LLM} \subset \mathsf{ReasoningModel} \subset \mathsf{ComputeHarness} \subset \mathsf{RecursiveHarness}$ in terms of capability, though not containment (a system is classified by its *primary* operating mode).

Only the third and fourth categories can:

- Notice incompleteness
- Allocate more compute
- Retry with new structure
- Escalate or abort

The fourth category additionally can:

- Spawn subordinate harnesses dynamically
- Restructure the execution graph at runtime
- Apply meta-level control over child orchestrators
- Enforce convergence criteria across recursive depth

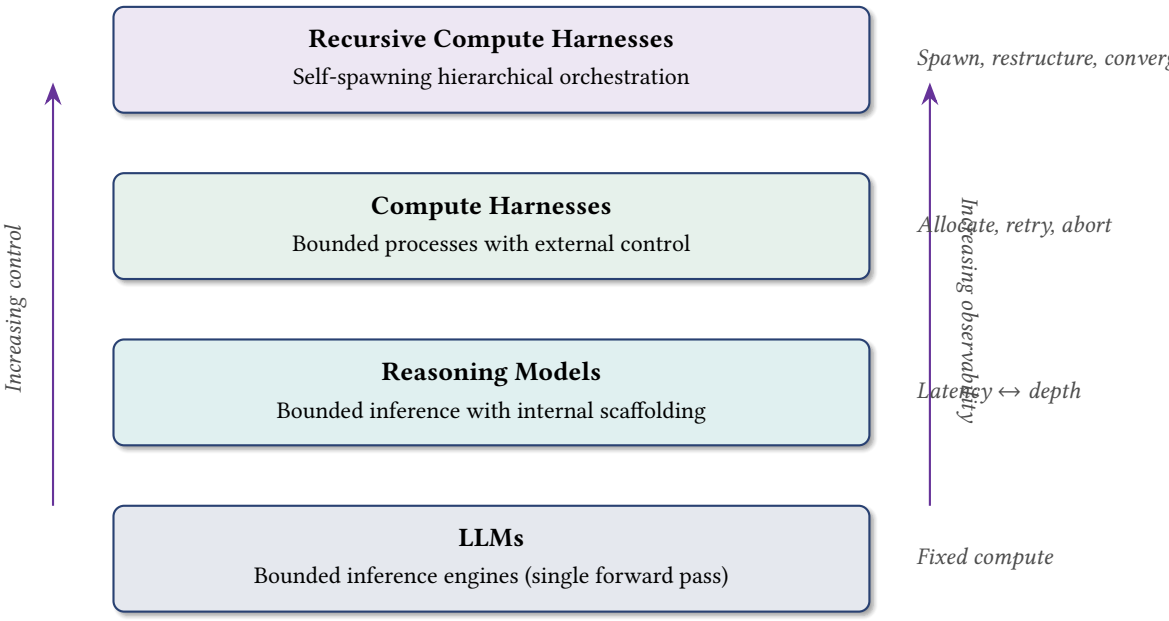

Fig. 4. The compute hierarchy: from single-pass inference to recursive orchestration.

16.5 The Incompleteness Theorem

Theorem 16.6 (Semantic-Compute Incompleteness). *Semantic incompleteness arises whenever the semantic obligations of a task exceed the compute budget allocated to its execution.*

$$\begin{aligned} \textit{Single forward pass} &\Rightarrow \textit{guaranteed incompleteness beyond trivial scope} \\ \textit{Reasoning model} &\Rightarrow \textit{delayed incompleteness} \\ \textit{Compute harness} &\Rightarrow \textit{visible incompleteness} \\ \textit{Recursive harness} &\Rightarrow \textit{managed incompleteness} \end{aligned}$$

Note on managed incompleteness: Recursive harnesses can detect when a subtask exceeds the capability of a child harness and restructure the decomposition—spawning additional children, increasing depth, or escalating to human oversight. Incompleteness is not eliminated but is made structurally addressable.

This is not an ML limitation. This is a computability constraint.

16.6 NPC as Incompleteness Management

The NPC architecture exists because:

- We do not trust any bounded computation to be complete
- We treat compute as a resource to allocate, not an oracle
- We insist on continuation, critique, and termination conditions

LLMs do not fail by being wrong.
They fail by being incomplete without knowing it.

16.7 The Mature Response

Semantic incompleteness is not a defect to eliminate. It is a fact to design around—like floating point error, packet loss, race conditions, or partial failure.

The failure mode changes from:

silent loss inside a forward pass

to:

explicit non-convergence over time

That is the difference between an impressive demo and a discipline.

16.8 Computational Complexity Analysis

We analyze the complexity of NPC execution across three dimensions: time, space, and communication.

Definition 16.7 (NPC Complexity Measures). For a neural program P executed by interpreter $\mathcal{I}$ in context Γ:

Time Complexity.

$$T(P) = \sum_{i=1}^{|P|} T_{\mathcal{I}}(s_i, \Gamma_i)$$

where s_i is the i-th statement and $T_{\mathcal{I}}$ is the interpreter's per-statement execution time. For transformer-based interpreters:

$$T_{\mathcal{I}}(s, \Gamma) = O(|\Gamma|^2 \cdot d)$$

where $|\Gamma|$ is context length in tokens and d is model dimension. This quadratic context dependence is the fundamental bottleneck.

Space Complexity.

$$S(P) = S_{\text{context}}(\Gamma) + S_{\text{activations}}(\mathcal{I}) + S_{\text{artifacts}}(P)$$

- $S_{\text{context}}(\Gamma) = O(|\Gamma| \cdot d)$: KV-cache for attention
- $S_{\text{activations}}(\mathcal{I}) = O(B \cdot n \cdot d)$: batch size B, sequence n
- $S_{\text{artifacts}}(P)$: domain-dependent (files, databases, etc.)

Communication Complexity. For programs with subagent delegation:

$$C(P) = \sum_{j=1}^{m} |\text{prompt}_j| + |\text{response}_j|$$

where m is the number of subagent invocations. Parallel subagents reduce wall-clock time but not total communication cost.

Proposition 16.8 (Complexity Bounds). *For manifestly terminating programs (Definition 9.6):*

(1) ***Sequential programs****:* $T(P) = O(|P| \cdot |\Gamma_{\max}|^2 \cdot d)$
(2) ***Bounded loops****:* $T(P) = O(k \cdot |P_{body}| \cdot |\Gamma_{\max}|^2 \cdot d)$ *for bound* k
(3) ***Parallel composition****:* $T(P_1 \| P_2) = \max(T(P_1), T(P_2))$ *wall-clock,* $T(P_1) + T(P_2)$ *compute*

Proposition 16.9 (Recursive Harness Complexity). *For recursive compute harnesses with maximum recursion depth* r*:*

(1) ***Time****:* $T(P) = O(r \cdot k \cdot |P| \cdot |\Gamma_{\max}|^2 \cdot d)$ *where* k *is the branching factor*
(2) ***Space****:* $S(P) = O(r \cdot S_{harness})$ *where* $S_{harness}$ *is per-level state*
(3) ***Communication****:* $C(P) = O(k^r \cdot (|prompt| + |response|))$ *in the worst case*

Recursive depth r trades compute for capability: deeper recursion enables more complex task decomposition but incurs multiplicative overhead. The branching factor k determines whether the system exhibits tree-like ($k > 1$) or chain-like ($k = 1$) execution structure.

Theorem 16.10 (Context Compression Tradeoff). *Applying the compression operator $\downarrow_r$ reduces time complexity:*

$$T(P, \downarrow_r (\Gamma)) = O(|P| \cdot (r \cdot |\Gamma|)^2 \cdot d)$$

at the cost of increased semantic loss (Theorem 7.1). The tradeoff is:

$$\textit{Speedup} = \frac{1}{r^2} \quad \textit{vs.} \quad \textit{Fidelity loss} \geq f(r)$$

where f is monotonically decreasing. Aggressive compression ($r \ll 1$) yields quadratic speedup but risks semantic incompleteness.

Definition 16.11 (Complexity Classes for Neural Programs). We define complexity classes relative to interpreter capability:

- NP-TRIVIAL: Solvable in single forward pass with high probability
- NP-BOUNDED: Solvable with bounded reasoning depth k
- NP-HARNESS: Requires external orchestration for reliable solution
- NP-RECURSIVE: Requires recursive orchestration with bounded depth r
- NP-UNDECIDABLE: No guaranteed termination (cf. Theorem 9.5)

Task classification is empirical and interpreter-dependent; these classes formalize observed capability boundaries rather than providing decision procedures. The NP-RECURSIVE class captures tasks requiring hierarchical decomposition that exceeds the coordination capacity of a single harness—such as large-scale document generation, multi-phase research synthesis, or recursive code refactoring.

17 NPC Extension: Claude Code Instantiation

This section provides a complete formal instantiation of the Neural Program Calculus for Claude Code, Anthropic's agentic coding assistant. Claude Code represents a production-grade realization of the NPC framework, with concrete tools, permission models, hook systems, and execution modes that require rigorous formalization.

17.1 Extended Semantic Domains

We extend the value domain $\mathcal{V}$ to include Claude Code-specific types:

Definition 17.1 (Extended Value Domain).

$$\begin{aligned}
\mathcal{V}_{\text{CC}} ::= \; & \mathcal{V} && \text{(base NPC values)} \\
| \; & \text{Tool}(t, \sigma) && \text{(tool with signature } \sigma) \\
| \; & \text{Permission}(p) && \text{(permission grant/deny)} \\
| \; & \text{HookResult}(h) && \text{(hook execution outcome)} \\
| \; & \text{ProcessHandle}(pid) && \text{(background process reference)} \\
| \; & \text{Session}(s) && \text{(conversation session state)} \\
| \; & \text{MCPResource}(r) && \text{(external MCP server resource)} \\
| \; & \text{SkillSpec}(k) && \text{(agent skill specification)}
\end{aligned}$$

Definition 17.2 (Extended Execution Context). The Claude Code execution context extends the base NPC context:

$$\Gamma_{\text{CC}} = \langle H, E, \rho, \kappa, \Pi, \Lambda, \Sigma, \mathcal{M} \rangle$$

where the additional components are:

- $\Pi : \text{Tool} \to \{\texttt{allow}, \texttt{deny}, \texttt{ask}\}$ is the permission mapping
- Λ is the set of registered hooks (event handlers)
- Σ is the session state (conversation history, checkpoints, settings)
- $\mathcal{M}$ is the set of connected MCP servers

17.2 Tool Taxonomy

Claude Code provides a fixed set of primitive tools, each with a formal signature.

Definition 17.3 (Tool Categories). The tool set $\mathcal{T}$ is partitioned into disjoint categories:

$$\begin{aligned}
\mathcal{T}_{\text{file}} &= \{\texttt{Read}, \texttt{Write}, \texttt{Edit}, \texttt{Glob}, \texttt{Grep}, \texttt{NotebookEdit}\} \\
\mathcal{T}_{\text{exec}} &= \{\texttt{Bash}, \texttt{KillShell}\} \\
\mathcal{T}_{\text{web}} &= \{\texttt{WebFetch}, \texttt{WebSearch}\} \\
\mathcal{T}_{\text{agent}} &= \{\texttt{Task}, \texttt{TaskOutput}, \texttt{Skill}\} \\
\mathcal{T}_{\text{interact}} &= \{\texttt{AskUserQuestion}, \texttt{TodoWrite}\} \\
\mathcal{T}_{\text{control}} &= \{\texttt{EnterPlanMode}, \texttt{ExitPlanMode}\}
\end{aligned}$$

The complete tool set is:

$$\mathcal{T} = \mathcal{T}_{\text{file}} \cup \mathcal{T}_{\text{exec}} \cup \mathcal{T}_{\text{web}} \cup \mathcal{T}_{\text{agent}} \cup \mathcal{T}_{\text{interact}} \cup \mathcal{T}_{\text{control}}$$

17.3 Tool Signatures

Each tool has a formal type signature. We use the notation $\tau_1 \xrightarrow{E} \tau_2$ to denote a function that may produce effects E during execution.

Definition 17.4 (File Operation Signatures).

$$\begin{aligned}
\texttt{Read} &: \text{Path} \times \text{Options} \xrightarrow{\text{IO}} \text{Content} \cup \{\bot\} \\
\texttt{Write} &: \text{Path} \times \text{Content} \xrightarrow{\text{IO,FS}} \text{Unit} \cup \{\bot\} \\
\texttt{Edit} &: \text{Path} \times \text{OldStr} \times \text{NewStr} \xrightarrow{\text{IO,FS}} \text{Unit} \cup \{\bot\} \\
\texttt{Glob} &: \text{Pattern} \times \text{Path?} \xrightarrow{\text{IO}} \text{List[Path]} \\
\texttt{Grep} &: \text{Regex} \times \text{Options} \xrightarrow{\text{IO}} \text{SearchResult} \\
\texttt{NotebookEdit} &: \text{Path} \times \text{CellSpec} \times \text{Content} \xrightarrow{\text{IO,FS}} \text{Unit} \cup \{\bot\}
\end{aligned}$$

where:

- Options includes offset, limit, and format parameters
- SearchResult = List[Path × Line? × Content?]
- Effect IO denotes filesystem read
- Effect FS denotes filesystem mutation

Definition 17.5 (Execution Tool Signatures).

$$\texttt{Bash} : \text{Command} \times \text{Options} \xrightarrow{\text{Proc,IO,Net?}} \text{Output} \cup \{\bot\}$$

$$\texttt{KillShell} : \text{ShellId} \xrightarrow{\text{Proc}} \text{Unit} \cup \{\bot\}$$

where:

- Options includes timeout, background flag, description
- Effect Proc denotes process creation/manipulation
- Effect Net conditionally present if command accesses network
- Background execution returns ProcessHandle instead of Output

Definition 17.6 (Web Tool Signatures).

$$\texttt{WebFetch} : \text{URL} \times \text{Prompt} \xrightarrow{\text{Net,LLM}} \text{Content} \cup \{\bot\}$$

$$\texttt{WebSearch} : \text{Query} \times \text{DomainFilters?} \xrightarrow{\text{Net}} \text{SearchResults}$$

where:

- Effect LLM denotes internal LLM processing of fetched content
- DomainFilters = AllowList ∪ BlockList

Definition 17.7 (Agent Tool Signatures).

$$\texttt{Task} : \text{Prompt} \times \text{AgentType} \times \text{Options} \xrightarrow{\text{Agent}} \text{Result} \cup \{\bot\}$$

$$\texttt{TaskOutput} : \text{TaskId} \times \text{BlockingMode} \xrightarrow{\emptyset} \text{Output} \cup \{\bot\}$$

$$\texttt{Skill} : \text{SkillName} \times \text{Args?} \xrightarrow{\text{Skill}} \text{Result}$$

where:

- Effect Agent denotes subagent spawning
- Effect Skill denotes skill invocation with its own effect set
- AgentType ∈ {`Bash`, `Explore`, `Plan`, `general-purpose`, custom}

Definition 17.8 (Interaction Tool Signatures).

$$\texttt{AskUserQuestion} : \text{Questions} \xrightarrow{\text{User}} \text{Answers}$$

$$\texttt{TodoWrite} : \text{TodoList} \xrightarrow{\text{State}} \text{Unit}$$

where:

- Effect User denotes blocking user interaction
- Effect State denotes internal state mutation
- Questions = List[Question × Options]

Definition 17.9 (Control Tool Signatures).

$$\texttt{EnterPlanMode} : \text{Unit} \xrightarrow{\text{Mode}} \text{Unit}$$

$$\texttt{ExitPlanMode} : \text{Unit} \xrightarrow{\text{Mode}} \text{Unit}$$

where Effect Mode denotes execution mode transition.

17.4 Extended Grammar

We extend the NPC grammar to include Claude Code-specific constructs:

Definition 17.10 (Extended Grammar). ⟨*stmt*⟩ ::= ... *(base NPC statements)*
| ⟨*tool-invocation*⟩
| ⟨*hook-definition*⟩
| ⟨*permission-block*⟩
| ⟨*mode-transition*⟩
| ⟨*mcp-invocation*⟩

⟨*tool-invocation*⟩ ::= ⟨*tool-name*⟩ '(' ⟨*tool-params*⟩ ')'

⟨*tool-name*⟩ ::= 'Read' | 'Write' | 'Edit' | 'Glob' | 'Grep' | 'Bash'
| 'WebFetch' | 'WebSearch' | 'Task' | 'Skill' | ...

⟨*tool-params*⟩ ::= ⟨*named-param*⟩ (',' ⟨*named-param*⟩)*

⟨*named-param*⟩ ::= ⟨*ident*⟩ '=' ⟨*expr*⟩

⟨*hook-definition*⟩ ::= 'on' ⟨*hook-event*⟩ ':' ⟨*hook-handler*⟩

⟨*hook-event*⟩ ::= 'PreToolUse' | 'PostToolUse' | 'PermissionRequest'
| 'SessionStart' | 'SessionEnd' | 'Stop' | 'SubagentStop'
| 'PreCompact' | 'Notification' | 'UserPromptSubmit'

⟨*hook-handler*⟩ ::= ⟨*block*⟩
| 'command' '(' ⟨*shell-command*⟩ ')'
| 'prompt' '(' ⟨*llm-prompt*⟩ ')'

⟨*permission-block*⟩ ::= 'with' 'permissions' ⟨*perm-spec*⟩ ':' ⟨*block*⟩

⟨*perm-spec*⟩ ::= '{' ⟨*perm-rule*⟩ (';' ⟨*perm-rule*⟩)* '}'

⟨*perm-rule*⟩ ::= ('allow' | 'deny' | 'ask') '(' ⟨*tool-pattern*⟩ ')'

⟨*mode-transition*⟩ ::= 'enter' ⟨*mode-name*⟩
| 'exit' ⟨*mode-name*⟩

⟨*mode-name*⟩ ::= 'plan' | 'interactive' | 'print' | 'background'

⟨*mcp-invocation*⟩ ::= 'mcp' '.' ⟨*server-name*⟩ '.' ⟨*tool-name*⟩ '(' ⟨*tool-params*⟩ ')'

17.5 Operational Semantics for Tools

Tool execution in Claude Code follows a three-phase protocol: permission check, hook interception, and actual execution.

Definition 17.11 (Tool Execution Protocol). For any tool invocation $t(\vec{p})$ in context Γ_{CC}:

$$\langle t(\vec{p}), \Gamma_{\text{CC}} \rangle \Downarrow_{\mathcal{I}} \mu$$

proceeds through three phases:

Phase 1: Permission Check

$$\text{CheckPerm}(t, \vec{p}, \Pi) = \begin{cases} \texttt{allow} & \text{if } \Pi(t, \vec{p}) = \texttt{allow} \\ \texttt{deny} & \text{if } \Pi(t, \vec{p}) = \texttt{deny} \\ \texttt{ask}(\text{user}) & \text{if } \Pi(t, \vec{p}) = \texttt{ask} \end{cases}$$

Phase 2: Hook Interception

$$\text{FireHooks}(\texttt{PreToolUse}, t, \vec{p}, \Lambda) \to (\vec{p}', d)$$

where $\vec{p}'$ is potentially modified parameters and $d \in \{\texttt{continue}, \texttt{block}\}$.

Phase 3: Execution

$$\text{if } d = \texttt{continue}: \quad \text{Execute}(t, \vec{p}', E) \to (v, E')$$

Phase 4: Post-Execution Hooks

$$\text{FireHooks}(\texttt{PostToolUse}, t, v, \Lambda) \to v'$$

The final result is:

$$\mu = \delta_{(v', \Gamma_{\text{CC}}[E \mapsto E'])}$$

where δ is the Dirac distribution.

Definition 17.12 (File Read Semantics).

$$\frac{\text{path} \in \text{dom}(E) \quad \Pi(\texttt{Read}, \text{path}) \neq \texttt{deny} \quad \text{content} = E(\text{path})[\text{offset} : \text{offset} + \text{limit}]}{\langle \texttt{Read}(\text{path}, \text{offset}, \text{limit}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\text{content}, \Gamma)}}$$

If the file does not exist:

$$\frac{\text{path} \notin \text{dom}(E)}{\langle \texttt{Read}(\text{path}, _, _), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\bot_{\text{FileNotFound}}, \Gamma)}}$$

Definition 17.13 (File Write Semantics).

$$\frac{\Pi(\texttt{Write}, \text{path}) = \texttt{allow} \quad E' = E[\text{path} \mapsto \text{content}]}{\langle \texttt{Write}(\text{path}, \text{content}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\text{Unit}, \Gamma[E \mapsto E'])}}$$

Definition 17.14 (Edit Semantics (Patch-Based)). The Edit tool performs exact string replacement:

$$\frac{\text{old} \in E(\text{path}) \quad \text{count}(\text{old}, E(\text{path})) = 1 \quad E' = E[\text{path} \mapsto E(\text{path})[\text{old}/\text{new}]]}{\langle \texttt{Edit}(\text{path}, \text{old}, \text{new}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\text{Unit}, \Gamma[E \mapsto E'])}}$$

If the old string is not unique:

$$\frac{\text{count}(\text{old}, E(\text{path})) \neq 1}{\langle \texttt{Edit}(\text{path}, \text{old}, \text{new}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\bot_{\text{NonUnique}}, \Gamma)}}$$

Definition 17.15 (Bash Execution Semantics). Bash execution creates a subprocess and captures output:

$$\frac{\Pi(\texttt{Bash}, \text{cmd}) = \texttt{allow} \quad (\text{stdout}, \text{stderr}, \text{exit}) = \text{exec}_{\text{shell}}(\text{cmd}, \text{timeout})}{\langle \texttt{Bash}(\text{cmd}, \text{timeout}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\text{BashResult}(\text{stdout}, \text{stderr}, \text{exit}), \Gamma)}}$$

For background execution:

$$\frac{\text{bg} = \texttt{true} \quad \text{pid} = \text{spawn}_{\text{shell}}(\text{cmd})}{\langle \texttt{Bash}(\text{cmd}, \text{bg} = \texttt{true}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(\text{ProcessHandle}(\text{pid}), \Gamma)}}$$

17.6 Hook System Semantics

The hook system provides interception points in the execution pipeline.

Definition 17.16 (Hook Events). The set of hook events $\mathcal{E}_{\text{hook}}$ is:

$$\begin{aligned}\mathcal{E}_{\text{hook}} = \{&\texttt{PreToolUse}, \texttt{PostToolUse}, \texttt{PermissionRequest},\\ &\texttt{SessionStart}, \texttt{SessionEnd}, \texttt{Stop}, \texttt{SubagentStop},\\ &\texttt{PreCompact}, \texttt{Notification}, \texttt{UserPromptSubmit}\}\end{aligned}$$

Definition 17.17 (Hook Configuration). A hook configuration $\lambda \in \Lambda$ is a tuple:

$$\lambda = \langle e, m, h, t \rangle$$

where:

- $e \in \mathcal{E}_{\text{hook}}$ is the triggering event
- m is a matcher (regex or tool pattern)
- h is the handler (command or prompt)
- t is the timeout in milliseconds

Definition 17.18 (Hook Execution Semantics). When event e fires with payload p:

$$\text{FireHooks}(e, p, \Lambda) = \begin{cases} (\text{transform}(p, r), \texttt{continue}) & \text{if } r.\text{decision} = \texttt{allow} \\ (p, \texttt{block}) & \text{if } r.\text{decision} = \texttt{block} \\ (p, \texttt{continue}) & \text{if no matching hooks} \end{cases}$$

where r is the result of executing all matching hooks:

$$r = \text{aggregate}(\{h(p) \mid \lambda = \langle e', m, h, t \rangle \in \Lambda, e' = e, m \text{ matches } p\})$$

Definition 17.19 (Command Hook Semantics). For a command hook with shell script s:

$$h_{\text{cmd}}(p) = \text{exec}_{\text{shell}}(s, \text{stdin} = \text{JSON}(p))$$

The result is interpreted as:

- Exit code 0: `allow`
- Exit code 2: `block` with reason from stderr
- Other exit codes: `allow` with warning

Definition 17.20 (Prompt Hook Semantics). For a prompt hook with LLM query q:

$$h_{\text{prompt}}(p) = \mathcal{I}_{\text{haiku}}(q, \text{context} = p)$$

The LLM response must conform to the schema:

$$\text{HookResponse} = \{\text{decision} : \texttt{approve} \mid \texttt{block}, \text{reason} : \text{String?}\}$$

17.7 Permission Model

Claude Code implements a layered permission model with pattern-based rules.

Definition 17.21 (Permission Rule). A permission rule π has the form:

$$\pi = \text{action}(\text{pattern})$$

where:

- action $\in \{\texttt{allow}, \texttt{deny}, \texttt{ask}\}$
- pattern = Tool(arg-pattern*)

Definition 17.22 (Pattern Matching). Pattern matching follows these rules:

$$\begin{aligned} \text{match}(*, s) &= \texttt{true} \quad \forall s \\ \text{match}(\text{prefix}*, s) &= s.\text{startsWith}(\text{prefix}) \\ \text{match}(\text{exact}, s) &= (s = \text{exact}) \end{aligned}$$

For tool invocations:

$$\text{match}(\texttt{Tool}(p_1, \ldots, p_n), \texttt{Tool}(a_1, \ldots, a_m)) = \bigwedge_{i=1}^{\min(n,m)} \text{match}(p_i, a_i)$$

Definition 17.23 (Permission Resolution). Given a permission mapping Π (ordered list of rules) and tool invocation $t(\vec{a})$:

$$\text{Resolve}(\Pi, t, \vec{a}) = \pi_i.\text{action}$$

where π_i is the first rule in Π such that $\text{match}(\pi_i.\text{pattern}, t(\vec{a})) = \texttt{true}$.

If no rule matches, the default is `ask`.

Definition 17.24 (Permission Modes). Claude Code supports five permission modes:

$$\begin{aligned} \text{Mode}_{\text{default}} &: \text{Standard permission checking} \\ \text{Mode}_{\text{acceptEdits}} &: \Pi' = \Pi[\forall t \in \mathcal{T}_{\text{file}} : t \mapsto \texttt{allow}] \\ \text{Mode}_{\text{dontAsk}} &: \Pi' = \Pi[\texttt{ask} \mapsto \texttt{allow}] \\ \text{Mode}_{\text{bypassPermissions}} &: \Pi' = \lambda t.\, \texttt{allow} \\ \text{Mode}_{\text{plan}} &: \Pi' = \Pi[\forall t \in \mathcal{T} \setminus \mathcal{T}_{\text{file}}^{\text{read}} : t \mapsto \texttt{deny}] \end{aligned}$$

17.8 Subagent Semantics

Claude Code supports multiple specialized subagent types.

Definition 17.25 (Subagent Types). The subagent type system $\mathcal{A}$ consists of:

$$\begin{aligned} \mathcal{A} = \{&\texttt{Bash}(\kappa_{\text{bash}}), \\ &\texttt{Explore}(\kappa_{\text{explore}}), \\ &\texttt{Plan}(\kappa_{\text{plan}}), \\ &\texttt{GeneralPurpose}(\kappa_{\text{all}}), \\ &\texttt{Custom}(n, \kappa_n)\} \end{aligned}$$

where κ denotes the capability set for each agent type.

Definition 17.26 (Agent Capability Sets).

$$\begin{aligned} \kappa_{\text{bash}} &= \{\texttt{Bash}\} \\ \kappa_{\text{explore}} &= \{\texttt{Read}, \texttt{Glob}, \texttt{Grep}, \texttt{Bash}|_{\text{readonly}}\} \\ \kappa_{\text{plan}} &= \{\texttt{Read}, \texttt{Glob}, \texttt{Grep}, \texttt{Bash}|_{\text{readonly}}\} \\ \kappa_{\text{all}} &= \mathcal{T} \end{aligned}$$

The restriction $\texttt{Bash}|_{\text{readonly}}$ limits commands to:

$$\texttt{Bash}|_{\text{readonly}} = \{\texttt{ls}, \texttt{git}, \texttt{find}, \texttt{cat}, \texttt{head}, \texttt{tail}\}$$

Definition 17.27 (Task Tool Semantics). The Task tool spawns a subagent with specified type:

$$\frac{\alpha \in \mathcal{A} \quad \Gamma_c = \text{derive}(\Gamma, \alpha.\kappa) \quad \langle \text{prompt}, \Gamma_c \rangle \Downarrow_{\mathcal{I}_\alpha} \mu_c}{\langle \texttt{Task}(\text{prompt}, \alpha), \Gamma \rangle \Downarrow_{\mathcal{I}} \text{merge}(\mu_c, \Gamma)}$$

where:

- $\text{derive}(\Gamma, \kappa')$ creates child context with capability intersection
- $\mathcal{I}_\alpha$ may use a different model (e.g., Haiku for Explore)
- Background execution (bg = `true`) returns immediately with task ID

Definition 17.28 (Agent Model Selection). Each agent type maps to a model tier:

$$\begin{aligned} \text{model}(\texttt{Explore}) &= \text{Haiku} \quad \text{(fast, lightweight)} \\ \text{model}(\texttt{Plan}) &= \text{Sonnet} \quad \text{(capable reasoning)} \\ \text{model}(\texttt{GeneralPurpose}) &= \text{Sonnet} \quad \text{(capable reasoning)} \\ \text{model}(\texttt{Custom}(n, _)) &= \text{config}(n).\text{model} \vee \text{inherit} \end{aligned}$$

17.9 MCP Integration

Model Context Protocol (MCP) extends the tool set with external servers.

Definition 17.29 (MCP Server). An MCP server $m \in \mathcal{M}$ is a tuple:

$$m = \langle \text{name}, \text{transport}, \text{tools}, \text{resources}, \text{prompts} \rangle$$

where:

- transport $\in \{\texttt{stdio}, \texttt{http}, \texttt{sse}\}$
- tools is the set of tools exposed by the server
- resources is the set of retrievable resources
- prompts is the set of prompt templates

Definition 17.30 (MCP Tool Invocation). MCP tools are invoked via namespaced syntax:

$$\texttt{mcp__server__tool}(\vec{p})$$

The operational semantics:

$$\frac{m \in \mathcal{M} \quad t \in m.\text{tools} \quad r = \text{rpc}(m, t, \vec{p})}{\langle \texttt{mcp__}m\texttt{__}t(\vec{p}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(r,\Gamma)}}$$

Definition 17.31 (MCP Resource References). Resources are referenced via @ syntax:

$$@\text{server} : \text{type} : //\text{identifier}$$

Resolution:

$$\text{resolve}(@m : t : //id) = \text{rpc}(m, \texttt{resource/read}, \{uri : t : //id\})$$

17.10 Session and Context Management

Definition 17.32 (Session State). A session Σ maintains:

$$\Sigma = \langle \text{id}, H, \text{checkpoints}, \text{settings}, \text{memory} \rangle$$

where:

- id is the unique session identifier
- H is the conversation history (messages and tool calls)
- checkpoints is the list of rewindable states
- settings is the merged configuration

- memory is the loaded CLAUDE.md content

Definition 17.33 (Context Compaction). The compression operator $\downarrow$ for Claude Code:

$$\downarrow(\Sigma) = \Sigma' = \langle \text{id}, H', \text{checkpoints}', \text{settings}, \text{memory} \rangle$$

where:

- $H' = \text{summarize}(H)$ with $|H'| < |H|$
- $\text{checkpoints}' = \text{last}(\text{checkpoints})$ (keep only most recent)

Definition 17.34 (Session Resume). Resuming a session:

$$\text{resume}(\text{id}) = \Gamma_{\text{CC}}[\Sigma \mapsto \text{load}(\text{id})]$$

The resumed context inherits:

- Full conversation history H
- All bindings ρ
- Original permission configuration Π

17.11 Execution Modes

Definition 17.35 (Execution Mode Transitions). Claude Code supports the following mode transitions:

$$\text{Interactive} \xrightarrow{\texttt{-p}} \text{Print}$$

$$\text{Interactive} \xrightarrow{\texttt{Shift+Tab}} \text{Plan}$$

$$\text{Plan} \xrightarrow{\texttt{ExitPlanMode}} \text{Interactive}$$

$$\text{Interactive} \xrightarrow{\texttt{Ctrl+B}} \text{Background}$$

Definition 17.36 (Plan Mode Semantics). In plan mode, the capability set is restricted:

$$\Gamma_{\text{plan}} = \Gamma_{\text{CC}}[\kappa \mapsto \kappa \cap \mathcal{T}^{\text{read}}]$$

where:

$$\mathcal{T}^{\text{read}} = \{\texttt{Read}, \texttt{Glob}, \texttt{Grep}, \texttt{Bash}|_{\text{readonly}}, \texttt{WebFetch}, \texttt{WebSearch}\}$$

No file mutations are permitted. The interpreter researches and produces a plan without modifying the codebase.

Definition 17.37 (Print Mode Semantics). Print mode (`-p` flag) executes non-interactively:

$$\text{print}(P, \text{input}) = \begin{cases} \text{JSON}(v) & \text{if output-format} = \texttt{json} \\ \text{stream}(\mu) & \text{if output-format} = \texttt{stream-json} \\ \text{text}(v) & \text{otherwise} \end{cases}$$

where $(v, _) \sim \mu$ from $\langle P, \Gamma_{\text{CC}} \rangle \Downarrow_I \mu$.

Print mode terminates after `max-turns` iterations.

17.12 Skill System

Definition 17.38 (Skill Specification). A skill k is defined by:

$$k = \langle \text{name}, \text{description}, \text{tools}, \text{model}, \text{context}, \text{hooks} \rangle$$

where:

- name is the unique identifier (invocable as /name)

- description triggers automatic invocation
- tools $\subseteq \mathcal{T}$ restricts available tools
- model overrides the default model
- context $\in \{\texttt{main}, \texttt{fork}\}$ determines isolation
- hooks defines skill-specific event handlers

Definition 17.39 (Skill Invocation Semantics). When a skill is invoked:

$$\frac{k \in \text{Skills} \quad \Gamma_k = \text{apply}(\Gamma, k.\text{tools}, k.\text{context}) \quad \langle k.\text{body}, \Gamma_k \rangle \Downarrow_{\mathcal{I}_{k.\text{model}}} \mu}{\langle \texttt{Skill}(k.\text{name}), \Gamma \rangle \Downarrow_{\mathcal{I}} \mu}$$

If k.context = `fork`, results are not merged back to parent.

17.13 Code Execution and Interpretation

A key capability of Claude Code is the ability to write, execute, and interpret code in an iterative development loop.

Definition 17.40 (Development Loop Operator). The development loop $\mathcal{D}$ encapsulates the write-run-debug cycle:

$$\mathcal{D}(P) = \texttt{while not } P.\texttt{validated} : \{\texttt{write}(P); \texttt{run}(P); \texttt{interpret}(P.\text{output}); \texttt{fix}(P)\}$$

This is the implicit structure of code generation tasks.

Definition 17.41 (Code Execution Semantics). When executing code:

$$\frac{\text{lang} = \text{detect}(\text{code}) \quad \text{cmd} = \text{runner}(\text{lang}, \text{code}) \quad (v, \Gamma') = [\![\texttt{Bash}(\text{cmd})]\!]_\Gamma}{\langle \texttt{run}(\text{code}), \Gamma \rangle \Downarrow_{\mathcal{I}} \delta_{(v,\Gamma')}}$$

where runner maps language to execution command:

$$\begin{aligned} \text{runner}(\texttt{python}, c) &= \texttt{python3 -c "}\text{escape}(c)\texttt{"} \\ \text{runner}(\texttt{javascript}, c) &= \texttt{node -e "}\text{escape}(c)\texttt{"} \\ \text{runner}(\texttt{typescript}, c) &= \texttt{npx ts-node -e "}\text{escape}(c)\texttt{"} \\ &\vdots \end{aligned}$$

Definition 17.42 (Output Interpretation). The interpreter $\mathcal{I}$ processes execution output:

$$\text{interpret}_{\mathcal{I}}(\text{output}) = \begin{cases} \text{Success}(\text{result}) & \text{if exit code} = 0 \wedge \text{no errors in output} \\ \text{Error}(\text{diagnosis}) & \text{if exit code} \neq 0 \vee \text{error patterns match} \\ \text{Warning}(\text{issues}) & \text{if warnings detected} \end{cases}$$

The diagnosis includes:

- Error type classification
- Line number extraction
- Suggested fixes (via $\mathcal{I}$)

17.14 Composition Theorems

Theorem 17.43 (Tool Composition). *For tools t_1, t_2 with signatures $\tau_1 \xrightarrow{E_1} \tau_2$ and $\tau_2 \xrightarrow{E_2} \tau_3$:*

$$t_2 \circ t_1 : \tau_1 \xrightarrow{E_1 \cup E_2} \tau_3$$

The composition is valid iff $\Pi(t_1) = \texttt{allow} \wedge \Pi(t_2) = \texttt{allow}$.

Theorem 17.44 (Hook Non-Interference). *For hooks λ_1, λ_2 with disjoint matchers:*

$$match(\lambda_1.m, p) \wedge match(\lambda_2.m, p) = \bot \implies order(\lambda_1, \lambda_2) \text{ does not affect } FireHooks(e, p, \Lambda)$$

Theorem 17.45 (Permission Monotonicity). *For permission modes $m_1 \preceq m_2$ (where $\preceq$ orders by restrictiveness):*

$$\Pi_{m_1}(t) = \texttt{allow} \implies \Pi_{m_2}(t) \in \{\texttt{allow}, \texttt{ask}\}$$

The ordering is: plan $\preceq$ default $\preceq$ acceptEdits $\preceq$ dontAsk $\preceq$ bypassPermissions.

Proposition 17.46 (Subagent Capability Restriction). *For any subagent invocation:*

$$\kappa_{child} \subseteq \kappa_{parent}$$

Capabilities can only decrease through delegation, never increase.

17.15 Complete Tool Reference

Table 2. Complete Claude Code Tool Reference

Tool	Category	Description
`Read`	File	Read file contents with offset/limit
`Write`	File	Create or overwrite file
`Edit`	File	Exact string replacement in file
`Glob`	File	Pattern-based file search
`Grep`	File	Regex content search with filtering
`NotebookEdit`	File	Modify Jupyter notebook cells
`Bash`	Exec	Execute shell commands
`KillShell`	Exec	Terminate background process
`WebFetch`	Web	Fetch and analyze URL content
`WebSearch`	Web	Web search with domain filters
`Task`	Agent	Spawn specialized subagent
`TaskOutput`	Agent	Retrieve background task output
`Skill`	Agent	Invoke agent skill
`AskUserQuestion`	Interact	Multi-choice user query
`TodoWrite`	Interact	Task list management
`EnterPlanMode`	Control	Switch to plan mode
`ExitPlanMode`	Control	Exit plan mode

17.16 Invariant Verification

We verify that Claude Code satisfies the formal invariants from Section 15.

Proposition 17.47 (Boundedness (Invariant I)). *Claude Code enforces boundedness through:*

- *Timeout parameters on all tool executions*
- `max-turns` *limit in print mode*
- *Explicit process termination via* `KillShell`

Proposition 17.48 (Explicit State (Invariant II)). *All state is externalized:*

- *Session state persisted to filesystem*
- *Environment variables require explicit* `CLAUDE_ENV_FILE`

- *No implicit memory between sessions*

Proposition 17.49 (Traceability (Invariant III)). *Every tool invocation is logged:*

- *Conversation transcript includes all tool calls*
- *Hook system provides interception points*
- *Session files provide complete audit trail*

Proposition 17.50 (Killability (Invariant X)). *All processes can be terminated:*

- `Ctrl+C` *interrupts active generation*
- `KillShell` *terminates background processes*
- *Session can be abandoned without corruption*

18 Conclusion

We have presented a preliminary formalization of the neural program paradigm—a notation for specifying complex tasks to self-organizing AI systems. The calculus combines formal syntax with probabilistic, intent-based semantics, reflecting the nature of its interpreters.

This work is necessarily incomplete. The semantics depend on properties of the interpreter $\mathcal{I}$ that are emergent rather than designed, making traditional proofs difficult. Nevertheless, we believe formalization is valuable: it clarifies what we mean, reveals implicit assumptions, and suggests avenues for both theoretical investigation and practical improvement.

Glushkov would, we hope, approve of the attempt—even if he would find much to criticize in the execution.

"To find the mathematics beneath the intuition,
to discover the theorems latent in the practice."

A Complete Examples

This appendix presents complete neural programs with annotations explaining their structure, execution semantics, and expected interpreter behavior.

A.1 Example 1: Document Translation (The Glushkov Case)

This is the neural program that initiated the translation of Glushkov's *Introduction to Cybernetics* from Markdown to LaTeX.

```
execute({
  sample( file.structure );

  for section in file.sections:
    subagent.write_latex(section)
    subagent.validate(completeness, accuracy, polish)

  assemble_final_latex( `${file}-complete.tex` );
  compile_latex_to_pdf( `${file}.pdf` )
})
```

Listing 10. Document Translation Neural Program

A.1.1 *Syntactic Analysis.*

Line	Construct	Type
1	`execute({...})`	Program wrapper
2	`sample(file.structure)`	Action (reconnaissance)
4	`for section in file.sections:`	Loop (parallelizable)
5	`subagent.write_latex(section)`	Delegated action
6	`subagent.validate(...)`	Delegated validation
8	`assemble_final_latex(...)`	Action (synthesis)
9	`compile_latex_to_pdf(...)`	Action (transformation)

A.1.2 *Execution Trace.*

(1) **Context Resolution**: The interpreter resolves `file` from conversation history H, binding it to the 657KB Markdown document.
(2) **Sampling**: `sample(file.structure)` triggers the interpreter to:
- Read portions of the file
- Identify structural patterns (chapters, sections, frontmatter, backmatter)
- Construct a mental model of document organization
- Bind `file.sections` to the identified components

(3) **Parallel Iteration**: The `for` loop spawns 8 parallel subagents (one per major section of the source document):
- Subagent 1: Frontmatter (title, abstract, preface)
- Subagent 2: Chapter I (Abstract Theory of Algorithms)
- Subagent 3: Chapter II (Boolean Functions)
- Subagent 4: Chapter III (Theory of Automata)
- Subagent 5: Chapter IV (Self-Organizing Systems)
- Subagent 6: Chapter V (Electronic Digital Machines)
- Subagent 7: Chapter VI (Predicate Calculus)
- Subagent 8: Backmatter (Bibliography, TOC)

(4) **Subagent Execution**: Each subagent independently:

- Reads its assigned section
- Transforms Markdown to LaTeX
- Self-validates against criteria
- Returns result to parent context

(5) **Assembly**: `assemble_final_latex` merges all outputs with:
- Unified preamble
- Proper `\input{}` directives or inline inclusion
- Consistent theorem environments
- Complete bibliography

(6) **Compilation**: `compile_latex_to_pdf` invokes the LaTeX toolchain.

A.1.3 Validation Semantics. The criteria `completeness, accuracy, polish` were interpreted as:

- `completeness`: Every section, equation, and reference in source appears in output
- `accuracy`: Mathematical notation is correctly rendered; no semantic distortion
- `polish`: Consistent formatting; proper theorem environments; clean typography

A.1.4 Result. Output: 117-page PDF, all 6 chapters, 79 bibliography entries, full table of contents.

A.2 Example 2: Code Review

A neural program for reviewing a pull request:

```
execute({
  fetch( pr.diff );
  fetch( pr.description );

  for file in pr.changed_files:
    analyze(file.changes)
    check(security_vulnerabilities)
    check(style_consistency)
    check(test_coverage)

  summarize_findings();

  if findings.severity > "warning":
    recommend(request_changes)
  else:
    recommend(approve)
})
```

Listing 11. Code Review Neural Program

A.2.1 Annotations.

- **Lines 2–3**: Implicit context—`pr` resolved from conversation (e.g., a GitHub URL mentioned earlier)
- **Line 5**: `pr.changed_files` is dynamically computed; the interpreter must fetch the PR diff and parse it
- **Lines 7–9**: `check(X)` is an abstract action; the interpreter determines *how* to check for security vulnerabilities, style, etc., based on:
  - File type (Python, JavaScript, Rust, etc.)
  - Project conventions (if detectable)
  - Known vulnerability patterns

- **Lines 12–15**: Conditional recommendation based on aggregated findings. The predicate `findings.severity > "warning"` is interpreted as an ordinal comparison (info < warning < error < critical).

A.3 Example 3: Data Pipeline

A neural program for ETL (Extract-Transform-Load):

```
execute({
  define clean(record) := {
    normalize(record.fields)
    validate(schema_conformance)
    if invalid: log_and_skip()
  }

  define enrich(record) := {
    lookup(record.user_id, external_api)
    merge(record, lookup_result)
  }

  stream source.records
    |> clean
    |> enrich
    |> write_to(destination)

  report(records_processed, records_skipped, errors)
})
```

Listing 12. Data Pipeline Neural Program

A.3.1 Annotations.

- **Lines 2–6, 8–11**: The `define` construct introduces reusable transformations. These are *not* executed immediately but bound as callable abstractions.
- **Lines 13–16**: The `stream ... |>` construct denotes a data pipeline:
  - Records flow from `source.records`
  - Each record passes through `clean`, then `enrich`
  - Valid records are written to `destination`
- **Implicit parallelism**: The interpreter may parallelize record processing if records are independent.
- **Line 18**: `report(...)` generates a summary using metrics accumulated during execution.

A.4 Example 4: Research Task with Error Handling

A neural program for investigating a technical question:

```
execute({
  clarify(question.scope, question.constraints);

  parallel for approach in [
    search(academic_papers),
    search(documentation),
    search(source_code),
    search(discussions)
```

```
  ]:
    try:
      subagent.investigate(approach)
      subagent.validate(relevance, credibility)
    on_failure:
      log(approach, "no results")
      continue

  synthesize(findings);

  if confidence < 0.7:
    identify_gaps();
    suggest_followup_queries()
  else:
    present_conclusions(with_citations)
})
```

Listing 13. Research Task Neural Program

A.4.1 Annotations.

- **Line 2**: `clarify` is a meta-action that may involve asking the user for clarification before proceeding.
- **Lines 4–8**: Explicit `parallel for` over heterogeneous search strategies.
- **Lines 10–15**: Error handling with `try/on_failure`. Failed searches do not abort the program; they are logged and skipped.
- **Lines 19–24**: Conditional logic based on self-assessed confidence. Low confidence triggers gap analysis; high confidence proceeds to conclusions.
- **Credibility validation**: The interpreter must assess source quality—peer-reviewed papers weighted higher than forum posts.

A.5 Example 5: Recursive Decomposition

A neural program demonstrating recursive task decomposition:

```
execute({
  define solve(problem) := {
    if is_atomic(problem):
      return direct_solution(problem)
    else:
      subproblems = decompose(problem)
      solutions = for p in subproblems: subagent.solve(p)
      return combine(solutions)
  }

  solve(original_problem);
  validate(correctness, optimality)
})
```

Listing 14. Recursive Decomposition Neural Program

A.5.1 Annotations.

- **Recursive definition**: `solve` calls itself via `subagent.solve(p)`.

- **Base case**: `is_atomic(problem)` determines when to stop recursing. The interpreter decides atomicity based on problem complexity.
- **Divide-and-conquer**: `decompose` splits a problem; `combine` merges sub-solutions.
- **Type signature** (implicit):

$$\texttt{solve} : \text{Problem} \to \text{Solution}$$

- **Parallel recursion**: Independent subproblems may be solved concurrently.

This pattern underlies many complex tasks: theorem proving, code generation, document processing, strategic planning.

A.6 Example 6: Multi-Stage Transformation with Checkpoints

A neural program for a complex multi-stage task with intermediate validation:

```
execute({
  // Stage 1: Analysis
  checkpoint("analysis") {
    parse(input);
    build_ast();
    analyze_dependencies();
    validate(parse_success, no_circular_deps)
  }

  // Stage 2: Transformation
  checkpoint("transform") {
    for node in ast.nodes:
      subagent.transform(node, target_format)
    validate(structural_preservation)
  }

  // Stage 3: Optimization
  checkpoint("optimize") {
    identify_redundancies();
    apply_simplifications();
    validate(semantic_equivalence)
  }

  // Stage 4: Output
  checkpoint("output") {
    serialize(result);
    write_to(destination);
    validate(roundtrip_consistency)
  }

  report(stages_completed, validations_passed, metrics)
})
```

Listing 15. Multi-Stage Transformation Neural Program

A.6.1 Annotations.

- **Checkpoints**: The `checkpoint(name) {...}` construct:
  - Names a stage for logging/debugging

  - Creates a restore point if later stages fail
  - Enables progress reporting
- **Stage isolation**: Each checkpoint has its own validation criteria. Failure at stage n does not lose work from stages 1 to $n-1$.
- **Comments**: `// ...` comments are ignored syntactically but may inform intent resolution.
- **Validation criteria by stage**:
  - Analysis: Parse success, no circular dependencies
  - Transform: Structural preservation (AST shape maintained)
  - Optimize: Semantic equivalence (meaning unchanged)
  - Output: Roundtrip consistency (can re-parse output)

A.7 Example 7: Human-in-the-Loop

A neural program that incorporates human decision points:

```
execute({
  draft(proposal);

  review_point("initial_review") {
    present(proposal);
    await human_feedback();

    if feedback.approved:
      proceed()
    else if feedback.revisions:
      incorporate(feedback.revisions);
      goto review_point("initial_review")
    else:
      abort("rejected by user")
  }

  implement(approved_proposal);

  review_point("final_review") {
    present(implementation);
    await human_approval();
  }

  finalize()
})
```

Listing 16. Human-in-the-Loop Neural Program

A.7.1 Annotations.

- **Blocking operations**: `await human_feedback()` suspends execution until the user responds.
- **Iterative refinement**: The `goto review_point` construct enables loops back to earlier stages (controlled, labeled goto—not arbitrary jumps).
- **Abort semantics**: `abort(reason)` terminates execution with an explanation.
- **Use case**: Design documents, code changes, or any task where human judgment is required at key decision points.

A.8 Summary of Examples

Example	Key Constructs	Domain
1. Document Translation	`for`, `subagent`, `validate`	Publishing
2. Code Review	`for`, `check`, `if`	Software Engineering
3. Data Pipeline	`define`, `stream`, `|>`	Data Engineering
4. Research Task	`parallel for`, `try/on_failure`	Research
5. Recursive Decomposition	`define`, recursion	General Problem Solving
6. Multi-Stage Transform	`checkpoint`	Compilers, Processing
7. Human-in-the-Loop	`await`, `review_point`	Collaborative Workflows

These examples demonstrate that NPC can express a wide variety of computational patterns while maintaining the essential characteristic: the interpreter resolves intent rather than executing literal instructions.

B Anti-Patterns

Each anti-pattern corresponds to one or more invariant violations. If you observe one of these in a system, the system is already compromised.

B.1 Anti-Pattern 1: Infinite Reflection

Violates: Invariant I (Boundedness), X (Killability)

Description: The system is allowed to "think again," "double-check," or "reflect" without a termination criterion.

Common disguises:

- "Let's reason step by step again"
- "Improve the answer iteratively"
- "Think harder"

Failure mode: Non-termination disguised as diligence.

Diagnosis rule: If you cannot state why the next step is the last step, the system is broken.

B.2 Anti-Pattern 2: Context as Memory

Violates: Invariant II (Explicit State), IX (Context Isolation)

Description: The system relies on prompt context as its memory substrate.

Common disguises:

- "It remembers because it's in the prompt"
- "The model keeps track internally"
- "We just pass the history"

Failure mode: Memory silently disappears, mutates, or bleeds across tasks.

Diagnosis rule: If deleting the prompt deletes the memory, it was never memory.

B.3 Anti-Pattern 3: Summary of a Summary

Violates: Invariant IV (Loss Accounting), III (Traceability)

Description: Outputs are repeatedly summarized without reference to original artifacts.

Common disguises:

- Rolling summaries
- "State compression"
- Recursive condensation

Failure mode: Semantic drift that feels coherent but is wrong.

Diagnosis rule: If you cannot reconstruct the original from references, loss is unaccounted.

B.4 Anti-Pattern 4: Top-K Truth

Violates: Invariant V (Identity Before Similarity), VI (Intent-Guided Retrieval)
Description: Similarity search is treated as relevance.
Common disguises:

- "The embedding was close"
- "This looked related"
- "Vector search returned it"

Failure mode: Important facts are buried while irrelevant ones dominate.
Diagnosis rule: If retrieval cannot explain *why* something was fetched, it was guessed.

B.5 Anti-Pattern 5: Generation Equals Decision

Violates: Invariant VII (Proposal ≠ Commitment)
Description: Model output is treated as authoritative by default.
Common disguises:

- Auto-execution
- "The model decided"
- "Agent autonomy"

Failure mode: Unchecked actions with no audit trail.
Diagnosis rule: If output can cause action without a gate, authority has leaked.

B.6 Anti-Pattern 6: Single-Path Confidence

Violates: Invariant VIII (Adversarial Pressure)
Description: The system accepts the first coherent answer.
Common disguises:

- "It looks right"
- "High confidence output"
- "No errors detected"

Failure mode: False convergence.
Diagnosis rule: If no independent critique path exists, correctness is untestable.

B.7 Anti-Pattern 7: Global Memory Pool

Violates: Invariant IX (Context Isolation)
Description: All tasks share a common memory store without strict scoping.
Common disguises:

- "Shared knowledge base"
- "Global agent memory"
- "Persistent context"

Failure mode: Cross-task contamination and nondeterminism.
Diagnosis rule: If a task can see data it did not explicitly request, isolation is broken.

B.8 Anti-Pattern 8: Unkillable Process

Violates: Invariant X (Killability)
Description: Processes that cannot be safely interrupted.
Common disguises:

- "Let it finish"
- "We don't want to corrupt state"
- "It's in the middle of reasoning"

Failure mode: Runaway systems and operator paralysis.
Diagnosis rule: If stopping it risks corruption, design already failed.

B.9 Anti-Pattern 9: Intelligence Metrics

Violates: Meta-Invariant
Description: Attempts to quantify "intelligence," "agency," or "autonomy."
Common disguises:

- Agent scores
- Reasoning depth metrics
- Autonomy levels

Failure mode: Misplaced optimization and self-mythology.
Diagnosis rule: If a system measures its own intelligence, it is lying.

B.10 Anti-Pattern 10: Magic Vocabulary

Violates: All invariants
Description: Language substitutes for structure.
Common disguises:

- "Emergent"
- "Self-aware"
- "Understands"
- "Reasons like a human"

Failure mode: Uninspectable systems protected by metaphor.
Diagnosis rule: If removing the word removes the explanation, it was never technical.

B.11 The Final Test

Can this system fail loudly, visibly, and early?

If the answer is no, it will fail silently, invisibly, and late.
That is the definition of an unserious system.

C Claude Code NPC Examples

This appendix presents complete NPC programs demonstrating Claude Code's capabilities in formal notation, with annotations explaining tool invocations, permission handling, hook interception, and subagent delegation.

C.1 Example 1: Codebase Exploration with Subagent Delegation

This example demonstrates the Explore subagent for fast codebase reconnaissance.

```
execute({
  // Delegate exploration to fast subagent
  structure = Task(
    prompt = "Find all TypeScript files implementing API endpoints",
    subagent_type = "Explore",
    model = "haiku"  // Fast, lightweight model
  );
```

```
  // Process results in main context
  for endpoint in structure.files:
    Read(file_path = endpoint);
    analyze(endpoint.handlers, endpoint.middleware);

  // Synthesize findings
  report(
    endpoints_found = len(structure.files),
    patterns_identified = structure.patterns
  )
})
```

Operational Trace:

(1) Task spawns Explore subagent with $\kappa_{\text{explore}} = \{\texttt{Read}, \texttt{Glob}, \texttt{Grep}\}$
(2) Subagent executes `Glob("**/*.ts")` followed by `Grep("@(Get|Post|Put|Delete)")`
(3) Results merged back to parent context via $\sqcup$
(4) Main context iterates over discovered files with full tool access

C.2 Example 2: Code Modification with Permission Handling

This example demonstrates the edit workflow with explicit permission checks.

```
execute({
  // Read target file (always permitted)
  content = Read(file_path = "/src/utils/helpers.ts");

  // Analyze and prepare modification
  old_code = extract_function(content, "formatDate");
  new_code = refactor(old_code, {
    use_intl_api: true,
    add_timezone_support: true
  });

  // Edit requires permission check
  // Permission rule: ask("Edit(src/*)")
  Edit(
    file_path = "/src/utils/helpers.ts",
    old_string = old_code,
    new_string = new_code
  );

  // Validate the change
  test_result = Bash(
    command = "npm test -- --grep formatDate",
    timeout = 60000
  );

  if test_result.exit_code != 0:
    // Rollback on failure
    Edit(
      file_path = "/src/utils/helpers.ts",
```

```
      old_string = new_code,
      new_string = old_code
    );
    report(status = "failed", reason = test_result.stderr)
  else:
    report(status = "success", tests_passed = test_result.stdout)
})
```

Permission Flow:

$$\Pi(\texttt{Read}, \texttt{"/src/..."}) = \texttt{allow} \quad \text{(read always permitted)}$$
$$\Pi(\texttt{Edit}, \texttt{"/src/..."}) = \texttt{ask} \quad \text{(user confirmation required)}$$
$$\Pi(\texttt{Bash}, \texttt{"npm test..."}) = \texttt{allow} \quad \text{(test commands permitted)}$$

C.3 Example 3: Hook-Driven Validation Pipeline

This example demonstrates the hook system for automated code quality checks.

```
// Hook configuration (in settings.json)
hooks = {
  PreToolUse: [
    {
      matcher: "Edit(*)",
      command: "./scripts/lint-check.sh",
      timeout: 10000
    }
  ],
  PostToolUse: [
    {
      matcher: "Write(*)",
      command: "./scripts/format-file.sh",
      timeout: 5000
    }
  ]
}

execute({
  // This Edit will trigger PreToolUse hook
  Edit(
    file_path = "/src/components/Button.tsx",
    old_string = "onClick={() => {}}",
    new_string = "onClick={handleClick}"
  );
  // Hook validates: eslint passes? -> continue : block

  // This Write will trigger PostToolUse hook
  Write(
    file_path = "/src/components/NewComponent.tsx",
    content = generate_component_template("NewComponent")
  );
  // Hook auto-formats with prettier
})
```

Hook Execution Semantics:

(1) `PreToolUse` fires before `Edit`
(2) Hook script receives JSON: `{tool: "Edit", file_path: "...", old_string: "...", new_string: "..."}`
(3) Exit code 0 → continue; Exit code 2 → block with message
(4) `PostToolUse` fires after `Write` completes
(5) Hook can modify file (formatter) without blocking

C.4 Example 4: MCP Integration for External Services

This example demonstrates MCP server integration for GitHub operations.

```
execute({
  // MCP server: github (configured via .mcp.json)

  // Fetch PR details via MCP tool
  pr_data = mcp__github__get_pull_request(
    owner = "anthropics",
    repo = "claude-code",
    pull_number = 123
  );

  // Get PR comments via MCP
  comments = mcp__github__list_comments(
    owner = "anthropics",
    repo = "claude-code",
    pull_number = 123
  );

  // Analyze the PR
  review = analyze_pr(pr_data, comments);

  // Post review comment via MCP
  mcp__github__create_review(
    owner = "anthropics",
    repo = "claude-code",
    pull_number = 123,
    body = review.summary,
    event = review.recommendation  // "APPROVE" | "REQUEST_CHANGES" | "COMMENT"
  )
})
```

MCP Tool Resolution:

$$\texttt{mcp__github__get_pull_request} \to \text{rpc}(m_{\text{github}}, \texttt{get_pull_request}, \vec{p})$$

The MCP server m_{github} handles authentication, rate limiting, and API translation.

C.5 Example 5: Multi-Agent Parallel Processing

This example demonstrates parallel subagent execution for complex tasks.

```
execute({
  // Discover all test files
```

```
  test_files = Glob(pattern = "**/*.test.ts");

  // Partition into batches for parallel execution
  batches = partition(test_files, batch_size = 10);

  // Launch parallel subagents
  results = parallel for batch in batches:
    Task(
      prompt = "Run tests and report failures: " + join(batch, ", "),
      subagent_type = "Bash",
      run_in_background = true
    );

  // Wait for all results
  all_results = for task_id in results:
    TaskOutput(task_id = task_id, block = true, timeout = 300000);

  // Aggregate results
  failures = flatten([r.failures for r in all_results]);

  if len(failures) > 0:
    // Fix failures sequentially (requires full context)
    for failure in failures:
      subagent.fix_test(failure);
      Bash(command = "npm test -- " + failure.file);

  report(
    total_tests = sum([r.total for r in all_results]),
    passed = sum([r.passed for r in all_results]),
    fixed = len(failures)
  )
})
```

Execution Model:

- `parallel for` spawns |batches| concurrent Bash subagents
- Each subagent has $\kappa_{\text{bash}} = \{\texttt{Bash}\}$
- `run_in_background = true` returns immediately with task ID
- `TaskOutput` with `block = true` waits for completion
- Sequential `subagent.fix_test` uses general-purpose agent with full capabilities

C.6 Example 6: Plan Mode Research

This example demonstrates plan mode for safe codebase analysis.

```
execute({
  // Enter plan mode (read-only)
  EnterPlanMode();

  // All mutations are blocked in plan mode
  // These operations are permitted:
  architecture = {
    entry_points: Glob(pattern = "**/index.{ts,tsx}"),
```

```
    config_files: Glob(pattern = "**/*.config.{js,ts,json}"),
    dependencies: Read(file_path = "package.json")
  };

  // Search for patterns
  api_routes = Grep(
    pattern = "@(Get|Post|Put|Delete|Patch)\\(",
    type = "ts"
  );

  // Fetch external documentation
  docs = WebFetch(
    url = "https://docs.example.com/api",
    prompt = "Extract API schema and authentication requirements"
  );

  // Synthesize plan (no code modifications)
  plan = synthesize_implementation_plan(
    architecture = architecture,
    api_routes = api_routes,
    external_docs = docs,
    target = "Add OAuth2 authentication"
  );

  // Present plan for approval
  present(plan);

  // Exit plan mode triggers user approval
  ExitPlanMode();

  // After approval, execute plan with full capabilities
  for step in plan.steps:
    execute_step(step)  // Now can Edit, Write, Bash, etc.
})
```

Capability Restriction in Plan Mode:

$$\kappa_{\text{plan}} = \{\texttt{Read}, \texttt{Glob}, \texttt{Grep}, \texttt{WebFetch}, \texttt{WebSearch}, \texttt{Bash}|_{\text{readonly}}\}$$

$$\kappa_{\text{plan}} \cap \{\texttt{Write}, \texttt{Edit}\} = \emptyset$$

C.7 Example 7: Skill Invocation with Context Forking

This example demonstrates skill invocation with isolated context.

```
// Skill definition (in .claude/skills/code-review.md)
// ---
// name: code-review
// description: Comprehensive code review with security analysis
// allowed-tools: Read, Glob, Grep, WebFetch
// context: fork
// model: sonnet
// ---
```

```
execute({
  // Get PR diff
  diff = Bash(command = "git diff main...HEAD");

  // Invoke skill in forked context
  // Skill cannot modify main context
  review = Skill(
    skill = "code-review",
    args = diff
  );

  // Review result is returned, but skill's internal
  // context (searches, reads, web lookups) is discarded

  // Apply review suggestions in main context
  for suggestion in review.suggestions:
    if suggestion.severity == "critical":
      Edit(
        file_path = suggestion.file,
        old_string = suggestion.old_code,
        new_string = suggestion.new_code
      )
})
```

Context Forking Semantics:

$$\Gamma_{\text{skill}} = \text{fork}(\Gamma) \implies \Gamma_{\text{skill}}.\text{bindings} \cap \Gamma.\text{bindings} = \emptyset$$

The forked context is discarded after skill completion; only the return value propagates.

C.8 Example 8: Complete Development Workflow

This example demonstrates a full development workflow combining all capabilities.

```
execute({
  // PHASE 1: Understand the task
  TodoWrite(todos = [
    { content: "Analyze existing auth system", status: "in_progress" },
    { content: "Design new OAuth2 flow", status: "pending" },
    { content: "Implement auth provider", status: "pending" },
    { content: "Add tests", status: "pending" },
    { content: "Update documentation", status: "pending" }
  ]);

  // PHASE 2: Research (using Explore subagent)
  auth_system = Task(
    prompt = "Find and analyze the current authentication implementation",
    subagent_type = "Explore",
    model = "haiku"
  );

  TodoWrite(todos = [
    { content: "Analyze existing auth system", status: "completed" },
    { content: "Design new OAuth2 flow", status: "in_progress" },
```

```
  ...
]);

// PHASE 3: Design (with user input)
design_choice = AskUserQuestion(
  questions = [{
    question: "Which OAuth2 provider should we integrate?",
    header: "Provider",
    options = [
      { label: "Google (Recommended)", description: "Most common, well-documented" },
      { label: "GitHub", description: "Good for developer tools" },
      { label: "Auth0", description: "Full-featured, enterprise-ready" }
    ],
    multiSelect = false
  }]
);

// PHASE 4: Implementation
TodoWrite(todos = [
  ...,
  { content: "Implement auth provider", status: "in_progress" },
  ...
]);

// Generate provider code
Write(
  file_path = "/src/auth/oauth-provider.ts",
  content = generate_oauth_provider(design_choice.provider)
);

// Generate configuration
Write(
  file_path = "/src/config/oauth.ts",
  content = generate_oauth_config(design_choice.provider)
);

// Update existing auth module
Edit(
  file_path = "/src/auth/index.ts",
  old_string = "export { BasicAuth }",
  new_string = "export { BasicAuth, OAuthProvider }"
);

TodoWrite(todos = [
  ...,
  { content: "Implement auth provider", status: "completed" },
  { content: "Add tests", status: "in_progress" },
  ...
]);

// PHASE 5: Testing
Write(
```

```
  file_path = "/src/auth/__tests__/oauth-provider.test.ts",
  content = generate_tests("oauth-provider")
);

test_result = Bash(
  command = "npm test -- src/auth/__tests__/oauth-provider.test.ts",
  timeout = 120000
);

if test_result.exit_code != 0:
  // Interpret and fix failures
  failures = parse_test_output(test_result.stderr);
  for failure in failures:
    fix = diagnose_and_fix(failure);
    Edit(
      file_path = failure.file,
      old_string = failure.code,
      new_string = fix.code
    );
  // Re-run tests
  Bash(command = "npm test -- src/auth/__tests__/oauth-provider.test.ts");

TodoWrite(todos = [
  ...,
  { content: "Add tests", status: "completed" },
  { content: "Update documentation", status: "in_progress" }
]);

// PHASE 6: Documentation
Edit(
  file_path = "/README.md",
  old_string = "## Authentication\n\nBasic auth only.",
  new_string = "## Authentication\n\nSupports Basic auth and OAuth2 (" +
               design_choice.provider + ").\n\nSee [OAuth Setup Guide](docs/oauth.md)
   ."
);

Write(
  file_path = "/docs/oauth.md",
  content = generate_documentation("oauth-setup", design_choice.provider)
);

TodoWrite(todos = [
  { content: "Analyze existing auth system", status: "completed" },
  { content: "Design new OAuth2 flow", status: "completed" },
  { content: "Implement auth provider", status: "completed" },
  { content: "Add tests", status: "completed" },
  { content: "Update documentation", status: "completed" }
]);

report(
  files_created = 4,
```

```
    files_modified = 2,
    tests_passing = true,
    ready_for_review = true
  )
})
```

Tool Invocation Count:

- `TodoWrite`: 5 (progress tracking)
- `Task`: 1 (Explore subagent)
- `AskUserQuestion`: 1 (design decision)
- `Write`: 4 (new files)
- `Edit`: 2 (modifications)
- `Bash`: 2 (test execution)

C.9 Example 9: Error Recovery with Checkpoints

This example demonstrates checkpoint-based recovery from failures.

```
execute({
  checkpoint("initial") {
    // Gather requirements
    requirements = analyze_issue(issue_id);
    validate(requirements.complete);
  }

  checkpoint("implementation") {
    // Attempt implementation
    try:
      for file in requirements.affected_files:
        content = Read(file_path = file);
        modified = apply_changes(content, requirements);
        Write(file_path = file, content = modified);
    on_failure:
      // Revert to checkpoint
      revert_to("initial");
      report(status = "failed", checkpoint = "implementation");
      abort()
  }

  checkpoint("validation") {
    // Run tests
    result = Bash(command = "npm test");
    if result.exit_code != 0:
      // Don't revert, attempt fix
      diagnose_and_fix(result.stderr);
      // Retry validation
      result = Bash(command = "npm test");
      if result.exit_code != 0:
        revert_to("implementation");
        report(status = "tests_failed", attempts = 2)
  }
```

```
  checkpoint("complete") {
    report(status = "success");
  }
})
```

Checkpoint Semantics:

$$\text{checkpoint}(n) = \Gamma_n = \langle H_n, E_n, \rho_n, \kappa_n, \ldots \rangle$$

$$\text{revert_to}(n) = \Gamma \mapsto \Gamma_n$$

Checkpoints capture the complete execution context, enabling rollback on failure.

Acknowledgments

The authors thank the reviewers for their constructive feedback. This work builds upon the foundation established by V.M. Glushkov and the intellectual precedent of SudoLang by Eric Elliott and Parallel Drive. We also acknowledge the broader community of practitioners developing neural program methodologies in production systems.

References

[1] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, et al. 2022. Constitutional AI: Harmlessness from AI Feedback. *arXiv preprint arXiv:2212.08073* (2022).

[2] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research* 20, 28 (2019), 1–6.

[3] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus A. Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A Probabilistic Programming Language. *Journal of Statistical Software* 76, 1 (2017).

[4] Harrison Chase. 2022. LangChain. GitHub repository. https://github.com/langchain-ai/langchain

[5] George Cybenko. 1989. Approximation by Superpositions of a Sigmoidal Function. *Mathematics of Control, Signals and Systems* 2, 4 (1989), 303–314.

[6] Eric Elliott and Parallel Drive. 2023. SudoLang: A Powerful Pseudocode Programming Language for LLMs. GitHub repository. https://github.com/paralleldrive/sudolang

[7] Victor M. Glushkov. 1964. *Introduction to Cybernetics (Vvedenie v kibernetiku).* Academy of Sciences of the Ukrainian SSR, Kiev.

[8] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Uncertainty in Artificial Intelligence (UAI).*

[9] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1–2 (2017), 1–119.

[10] Geoffrey E. Hinton and Ruslan R. Salakhutdinov. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313, 5786 (2006), 504–507.

[11] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, et al. 2022. Training Compute-Optimal Large Language Models. *arXiv preprint arXiv:2203.15556* (2022).

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Addison-Wesley.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. 1989. Multilayer Feedforward Networks are Universal Approximators. *Neural Networks* 2, 5 (1989), 359–366.

[14] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12 (2023).

[15] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling Laws for Neural Language Models. *arXiv preprint arXiv:2001.08361* (2020).

[16] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Mober, Prayag Das, Hima Ramamurthy, and Matei Zaharia. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714* (2023).

[17] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models are Zero-Shot Reasoners. *Advances in Neural Information Processing Systems* 35 (2022).
[18] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep Learning. *Nature* 521, 7553 (2015), 436–444.
[19] Joon Sung Park, Joseph C. O'Brien, Carrie J. Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. 2023. Generative Agents: Interactive Simulacra of Human Behavior. *arXiv preprint arXiv:2304.03442* (2023).
[20] Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.
[21] Gordon D. Plotkin. 1977. LCF Considered as a Programming Language. *Theoretical Computer Science* 5, 3 (1977), 223–255.
[22] Jürgen Schmidhuber. 2015. Deep Learning in Neural Networks: An Overview. *Neural Networks* 61 (2015), 85–117.
[23] Dana Scott and Christopher Strachey. 1971. *Toward a Mathematical Semantics for Computer Languages.* Polytechnic Institute of Brooklyn. 19–46 pages.
[24] Claude E. Shannon. 1948. A Mathematical Theory of Communication. *Bell System Technical Journal* 27 (1948), 379–423, 623–656.
[25] Claude E. Shannon. 1959. Coding Theorems for a Discrete Source with a Fidelity Criterion. *IRE National Convention Record* 7 (1959), 142–163.
[26] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. *SNAPL: Summit on Advances in Programming Languages* (2015).
[27] Michael Sipser. 2012. *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.
[28] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph. D. Dissertation. University of California, Berkeley.
[29] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Advances in Neural Information Processing Systems*, Vol. 30.
[30] John von Neumann. 1945. *First Draft of a Report on the EDVAC.* Technical Report. University of Pennsylvania, Moore School of Electrical Engineering. Contract No. W-670-ORD-4926.
[31] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-Consistency Improves Chain of Thought Reasoning in Language Models. In *International Conference on Learning Representations (ICLR).*
[32] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, Vol. 35.
[33] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press.
[34] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR).*
[35] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. Tree of Thoughts: Deliberate Problem Solving with Large Language Models. In *Advances in Neural Information Processing Systems.*